AD-A254 568

2

REPORT

Form Approved
OMB No. 0704-0188

Public reporting burden for this collectio he time for reviewing instructions, searching existing data sources,
gathering and maintaining the data neec imments regarding this burden estimate or any other aspect of this
collection of information, including sugg directorate for information Operations and Reports, 1215 Jefferson
Davis Highway, Suite 1204, Arlington, VA leduction Project (0704-0188) Washington DC 20503

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
FINAL 1 Sep 88 - 31 Aug 91

4. TITLE AND SUBTITLE
"PLANNING AND CONTROL" (U)

5. FUNDING NUMBERS
6426/00
62301E

6. AUTHOR(S)
Professor Thomas Dean

DTIC
ELECTE
JUL 2 7 1992
S C D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Brown University
Department of Computer Science
Providence RI 02912

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 92 0686

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-88-C-0132

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
Distribution unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT *(Maximum 200 words)*

The research was devoted to the design of complex systems for applications in robotics, automated manufacturing, and time-critical decision support systems. In exploring the issues involved in the design of such systems, they investigated techniques from artificial intelligence, control theory, operations research, and the decision sciences. In the process, they attempted to draw correspondences between concepts from the various fields. However, this work was not intended as a grand unification of these disciplines, even as they pertain to the specific issues of interest. Instead, they presented tools from these areas as component technologies, each playing a pivotal role in the design of complex autonomous systems.

92 7 28 076

92-19946

14. SUBJECT TERMS

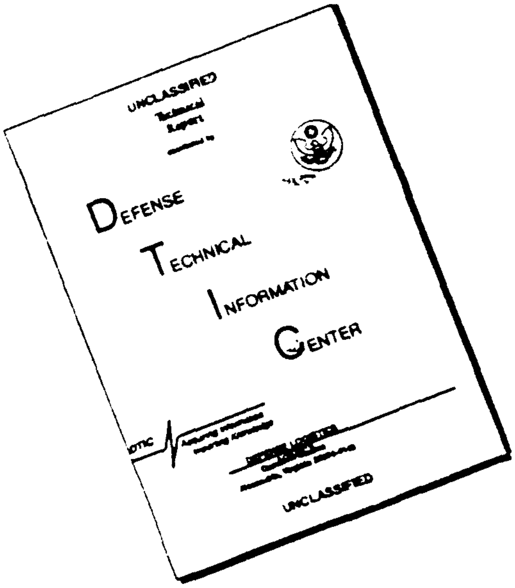
411436 391 pg

15. NUMBER OF PAGES
360

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

DISCLAIMER NOTICE



UNCLASSIFIED

Technical Report

DEFENSE

TECHNICAL

INFORMATION

CENTER

DTIC

UNCLASSIFIED

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Accession For
NTIS GRA&I ☒
DTIC TAB □
Unannounced □
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC QUALITY INSPECTED 3

Final Report

ARPA Order:	6426
Program Code:	8E20
Contractor:	Brown University
Effective Date:	September 1, 1988
Expiration Date:	August 31, 1991
Amount:	$440,000
Contract Number:	F49620-88-C-0132
Principal Investigator:	Thomas Dean (401) 863-7645
Program Manager:	Abraham Waksman (202) 767-5028
Title:	Coordinating Planning and Control

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 6426
Monitored by AFOSR Under
Contract No. F49620-88-C-0132

Planning and Control

Thomas L. Dean
Brown University

Michael P. Wellman
USAF Wright Laboratory

Morgan Kaufmann Publishers, Inc.

Preface

This book is devoted to the design of complex systems for applications in robotics, automated manufacturing, and time-critical decision support systems. In exploring the issues involved in the design of such systems, we investigate techniques from artificial intelligence, control theory, operations research, and the decision sciences. In the process, we attempt to draw correspondences between concepts from the various fields. However, this work is not intended as a grand unification of these disciplines, even as they pertain to the specific issues of interest. Instead, we present tools from these areas as component technologies, each playing a pivotal role in the design of complex autonomous systems.

In our attempt to draw a coherent picture of the broad range of problems and techniques considered here, we rely on the central themes of observation, prediction, and computation. In an uncertain environment, we must employ *observation* to augment our incomplete knowledge with evidence from the senses. We invoke *prediction* to extrapolate from our knowledge and observations the effects of our actions over time. Revising and making effective use of our knowledge requires *computation* to translate models and observations to meaningful action. The design of a system to control complex processes consists largely of strategies for deciding dynamically what and how to observe, predict, and compute.

In the 1980s, the traditional view of planning as offline computation relying on precise models and perfect information was challenged by research in artificial intelligence on robotic control systems embedded in complex environments. The challenge was met with proposals for *reactive systems*: systems designed to respond directly to perceived conditions in situations where there is little or no time to deliberate on how best to act. One disconcerting aspect of the focus on reactive systems was that it diverted effort from *planning*: predicting possible futures and formulating plans of action that take into account those possibilities. As research progressed, it became apparent that there was significant overlap between the work on reactive systems and the work in control theory. This book connects traditional research in planning with the constraints governing embedded systems, by reformulating the process of planning in terms of control.

Viewed from a control perspective, reactive systems embody particular strategies for controlling processes. In order to evaluate reactive systems, we have to analyze the connection between such strategies and the physical systems they seek to control. The tools required to perform such analyses are readily available from control theory, computer science, and artificial intelligence. This book focuses on the issues involved in modeling processes and generating sequences of commands in a timely manner. The practice of constructing formal models of physical systems and then using those models to develop programs to control processes is examined in some depth.

This book is intended for graduate and advanced undergraduate students in computer science and engineering. It is meant for students trying to orient themselves with respect to the many disciplines that have something significant to say about planning and control for applications in robotics and automation. The material in this book is suitable for a one-semester course offered to graduate and advanced undergraduate students. Given that the material covers a range of disciplines, we assume a somewhat varied background.

From computer science, we assume some familiarity with the theory of computation [12] and basic complexity theory [8]. Pidgin ALGOL [1] and Edinburgh PROLOG [5] are employed in describing algorithms. Some background in logic [14] and its application in artificial intelligence are also expected [4, 15]. Elementary probability theory plays a role in the chapters on uncertainty and stochastic modeling [11, 13]. While no background in control theory is required, we assume some familiarity with linear algebra and elementary differential equations [17]. We refer occasionally to standard techniques in robotics and machine vision, but no detailed knowledge is assumed. References, both general and specific, are provided at the end of each chapter, so that readers can fill in any missing background knowledge.

The book introduces advanced techniques that derive from work in a number of disciplines. The exposition of these techniques is largely self-contained, with pointers to more detailed treatments. In particular, the text explores the use of default reasoning [9] and temporal logics [18] in modeling processes, a framework for integrating techniques from control theory [6, 10] into a theory of planning, and several methods for coping with uncertainty derived from work in artificial intelligence [16], control theory [2], and decision analysis [3]. The phrase "Intelligent Control" was coined by Fu [7] to describe the field corresponding to the intersection of artificial intelligence and automatic control. Our interests in this book often coincide with those of the intelligent control community, and, where appropriate, we provide pointers to this literature.

The original idea for this book came from a course on robot problem-solving taught by Tom Dean at Brown University. In the Spring of 1989, Dean began work on a textbook based on his lecture notes for this course. Mike Wellman joined the project in the Fall of 1990. The collaboration has worked out well, and we expect to continue working together on future projects.

We consider this book as a tentative first step towards an integrated view of planning and control. We expect that the ideas presented herein will undergo major revision as the field proceeds to define itself. There were times when we began exploring details that threatened to delay the book by months if not years. Our editors, colleagues, and students persuaded us, however, that it was more important to publish a first approximation to the theory we were seeking in order to enlist the combined efforts of the rest of the research community. In the end, we were content to provide a rather high-level travel guide to exploring the territory. It is our hope and expectation that this book will be rewritten every three or four years for the foreseeable future; not necessarily by us, but by our students and colleagues in a variety of disciplines.

Acknowledgments

We would like to thank Steve Cross, Bob Simpson, and Rand Waltzman at DARPA, Abe Waksman at AFOSR, Nort Fowler at Rome Laboratory, Sanjaya Addanki at IBM, and Ken Laws at NSF for their support of the research that went into making this book possible.

A large portion of this book was completed while Tom Dean was spending his sabbatical leave at Stanford University. Jean-Claude Latombe and Yoav Shoham at Stanford provided a stimulating working environment that significantly influenced the final content of the book. Wonyun Choi—a graduate student at Stanford—and Hideki Isozaki—a visiting scholar from Japan—helped with some of the examples. Obtaining the early (junior) sabbatical leave would not have been possible without the support of Eugene Charniak and John Savage from Brown.

During the two years of preparation, we received encouragement and feedback from several people. Nils Nilsson of Stanford was enthusiastic about the book from its conception. We received useful feedback from Chris Brown at Rochester, Jon Doyle at MIT, Greg Hager and Dan Koditschek at Yale, Peter Ramadge and Elisha Sacks at Princeton, Stuart Russell at Berkeley, Reid Simmons at Carnegie-Mellon, Rich Sutton at GTE Laboratories, and Bill Wolovich at Brown. Many Brown students—Ken Basye, Mark Boddy, Ted Camus, Robert Chekaluk, Seungseok Hyun, Jak Kirman, Keiji Kanazawa, Jin Joo Lee, Moises Lejter, Neal Lesh, Oded Maron, Linda Nunez-Mensinger, Margaret Randazza, John Shewchuk, and Tu-Hsin Tsai—contributed to the research that went into this project. We are especially grateful to Neal Lesh and Oded Maron, who wrote the code for most of the examples used in the text. Finally, we wish to thank Mike Morgan for obtaining reviews in a timely manner, urging us to focus the book, and generally remaining enthusiastic and supportive throughout the whole project.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[2] M. J. Ashworth. *Feedback Design of Systems with Significant Uncertainty.* John Wiley and Sons, New York, 1982.

[3] V. Barnett. *Comparative Statistical Inference.* John Wiley and Sons, New York, 1982.

[4] Eugene Charniak and Drew V. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, 1985.

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1984.

[6] Richard C. Dorf. *Modern Control Systems.* Addison-Wesley, Reading, Massachusetts, 1989.

[7] K. S. Fu. Learning control systems and intelligent control systems: An intersection of artificial intelligence and automatic control. *IEEE Transactions on Automatic Control*, 16(1):70–72, 1971.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[9] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan-Kaufmann, Los Altos, California, 1987.

[10] Francis J. Hale. *Introduction to Control System Analysis and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[11] Paul G. Hoel, Sidney C. Port, and Charles J. Stone. *Introduction to Probability Theory.* Houghton Mifflin, Boston, Massachusetts, 1971.

[12] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[13] D. V. Lindley. *Introduction to Probability and Statistics.* Cambridge University Press, 1980.

[14] Elliot Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand, New York, 1979.

[15] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, California, 1980.

[16] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan-Kaufmann, Los Altos, California, 1988.

[17] Albert L. Rabenstein. *Elementary Differential Equations with Linear Algebra.* Academic Press, New York, 1975.

[18] Yoav Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence.* MIT Press, Cambridge, Massachusetts, 1988.

Chapter 1

Introduction

It is late and you are returning home after shoppi~~n~~g at the grocery store. You thread your car through the narrow streets of your neighborhood, and maneuver carefully into a parking place barely large enough to accommodate your vehicle. You gather up the groceries, walk up the steps to your apartment, and grope your way down the hall trying to feel the light switch so you can find the right key. After setting the groceries on the kitchen table, you put some leftovers in the oven, and step into the bathroom to start running a hot bath. Returning to the kitchen, you begin putting the groceries away. About midway through shelving the groceries, you return to the bathroom and adjust the faucets to ensure a comfortable temperature for your bath. When you return to the kitchen, you turn the oven down before finishing with the groceries.

Parking a car, carrying groceries, heating food, and running a warm bath are all examples of controlling processes. Quite often, we are engaged in controlling several processes simultaneously, as in the case of running a bath and heating leftovers. There are some processes that we have considerable control over, such as those having to do with the movement of our arms and legs, and other processes that we have very little control over, such as the process governing how many people in an apartment building are using the hot water at any given moment. There are limits, however, even to our control over our arms and legs. The arms and legs in conjunction with neural circuits in the spinal cord respond to stimuli without conscious effort: the arm jerks the hand back from a hot surface, the legs move involuntarily to save us from falling if we stumble. Many of the processes that we are used to

[0]©1990 Thomas Dean. All rights reserved.

dealing with on a day-to-day basis (*e.g.*, the weather) are completely outside of our control. We learn how to influence those processes we can exert some control over, and adapt our behavior to cope with those we cannot.

This ~~monograph~~ is concerned with the design of programs that control the behavior of physical processes. Intuitively, a process is just a series of changes in the state of the world. Controlling a process consists of making certain changes in the state of the world in order to determine *what* additional changes in the state of the world will occur and *when*. We distinguish between the *controller*, a device that includes hardware to run a control program, and the *controlled process*, often another device or group of devices whose behavior the controller is seeking to influence. In control theory, the controlled process is referred to as the *plant*. In robotics, the controlled process might correspond to certain mechanical components of the robot such as a manipulator or a drive mechanism, or it might correspond to the environment in which the robot is meant to function. The controller exerts control over the controlled process and monitors its progress through the use of auxiliary interface devices. Generally, these devices correspond to sensors and robotic manipulators, but there are other sorts of interfaces. For instance, the designer of a special-purpose microprocessor may view the microprocessor as the controller and its input and output ports as interface devices.

The distinction between controller and controlled process is quite natural from an engineer's point of view; the controller is a device that the engineer designs and builds. It is important to keep in mind, however, that the controller is itself a process. Both the controller and the controlled process operate in the same spatial and temporal context: both are embedded in a larger process. The study of control is the study of the relationship between controlling and controlled processes. This relationship is central to our investigations.

In order to control the behavior of a process, it is often useful to have some information concerning its current state. This information can be obtained in two different ways: you can observe the state directly, or you can predict it from information about earlier states. In order to predict the current state of a process from its past states, it is necessary to have a *model* of that process. A model is a description of a process used to derive information about present and future states of the process given information about its current and past states.

If you see a projectile hurtling toward you, then you might predict that the projectile will hit you if you remain in your current position, and you might use the prediction as a justification for your ducking. If you know that there is a protective barrier between you and the projectile, or you know that the projectile is tethered on a short string, then you can save yourself the trouble of ducking. Determining how to act to satisfy certain goals based upon predictions of possible future states is what is generally referred tc as *planning*. There are situations, however, in which making careful predictions is either unnecessary, impractical, or impossible.

When you place leftovers in an oven set at a certain temperature, you employ a very simple model to predict when those leftovers will be ready to eat. You could place a temperature sensor in the leftovers, and continually check the sensor until it reached a preset value. This is what is referred to as *monitoring* a process. Given the predictability of most ovens, it is hardly necessary to monitor the warming of leftovers. There are processes that are so unpredictable that they warrant constant monitoring (*e.g.*, air traffic over a metropolitan area). The decision of whether to monitor or predict the behavior of a process is a complex one involving subtle tradeoffs. Deploying sensors for monitoring can be expensive in that the sensors may not be available to monitor other processes. There are also often significant computational costs associated with both monitoring and prediction. The study of control is intimately tied up with utilizing scarce resources corresponding to sensors, manipulators, and associated computing machinery. Planning provides a framework for reasoning about tradeoffs and directly addresses the problem of resource utilization. This ~~monograph~~ explores control from the perspective of planning, and planning from the perspective of control. The idea being that the two are intimately related but emphasize different aspects of the same problem.

✓ book

In the rest of this chapter, we explore the notion of control and how it relates to planning in somewhat more detail. Our discussion will revolve around the idea of modeling processes and using models to direct control.

1.1 Controlling Processes

So far, we have talked about processes as though they actually exist in the world, whereas, in point of fact, they exist in our heads for the purpose of explaining our observations of physical phenomena. A process is an abstract description of physical phenomena. Such a description makes use of some vocabulary for speaking about the state of the world. For instance, we may want to speak about the position (x, y, and z coordinates) of a robot with

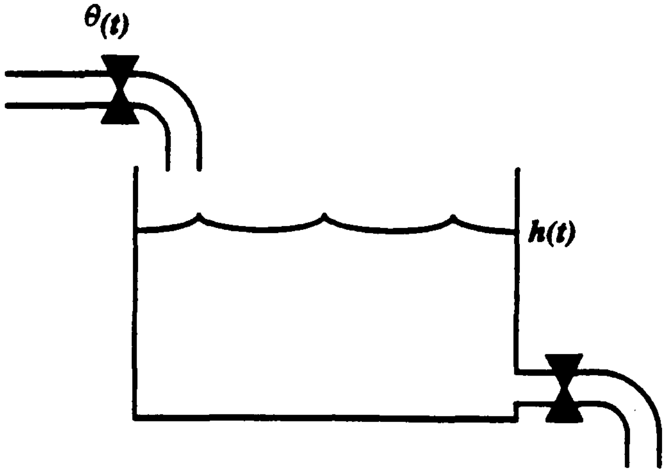


Figure 1.1: A simple control problem

respect to some frame of reference, or the charge (c measured in ampere hours) on a battery used to power the robot. Variables such as x, y, z, and c are referred to as *state variables*. We assume that the state of the world can be accurately described in terms of some number of state variables. Of course, the notion of accuracy has to be defined with respect to a particular task. Which brings us to an important question. Why do we want to describe the state of the world at all?

Presumably, we are interested in controlling (*i.e.*, influencing the value of) certain state variables. We are interested in other state variables insofar as they provide us with information that enables us to exercise better control. An example should make the discussion more concrete.

Figure 1.1 depicts a cylindrical tank containing fluid with one pipe leading in and one pipe leading out. There is a rotary valve mounted on each pipe that restricts the flow of fluid through the pipe. The position, θ, of the valve leading in determines how much fluid flows into the tank. In this example, we are interested in maintaining the height, h, of the fluid in the tank as close as possible to some preset value, say 3 meters, referred to as the ***target value***. We will assume that the valve mounted on the pipe leading out is locked in position.

The process that we are interested in controlling can be described by the two functions of time, $\theta(t)$ and $h(t)$, corresponding to the two state variables, θ and h. As far as we are concerned, the state of the world at a particular time t is determined by $\theta(t)$, and $h(t)$. We can predict future states of the process from past states if we have an appropriate model. For the process

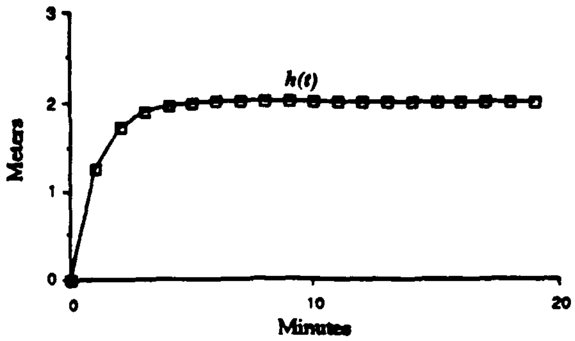


Figure 1.2: Change in fluid height for a constant valve position of 10°

described above, a simple first-order differential equation provides a suitable model,

$$K_{in}\theta(t) - K_{out}h(t) = A\frac{dh(t)}{dt},$$

where K_{in} is the [illegible] flow constant in cubic meters per degree minute for the valve governing flow through the input pipe. K_{out} is the [illegible] flow constant in square meters per minute for the output pipe, and A is the surface area of the tank. By solving this equation, we can predict the state of the process at time t, given information about the state of the process at some earlier time t_0. The solution to the above differential equation is

$$h(t) = Ce^{-(t\frac{K_{out}}{A})} + \theta(t_0)\frac{K_{in}}{K_{out}},$$

where C is obtained from the initial conditions as,

$$C = h(t_0) - \frac{K_{in}}{K_{out}}\theta(t_0).$$

Figure 1.2 shows the predictions made by the above model for a constant valve position of 10°, where $\frac{K_{in}}{K_{out}} = 0.2$ meters/degree, $\frac{K_{out}}{A} = 1$ minute, and the tank is initially empty. Note that if we are aware of changes in the variable θ, we can use this information and our model to make predictions about changes in the variable h. Given a sequence of changes in θ, we can evaluate the effectiveness of that sequence using the predicted changes in h and some set of criteria for effective control (*e.g.*, how rapidly h converges to the target value).

We still need to specify how the controller senses the world and how it might act to control the height of the fluid in the tank. Figure 1.3 depicts the two sensors used by the controller: one that provides information about h, and a second that provides information about θ. In addition, we will assume that the controller can influence θ by issuing one of two commands:

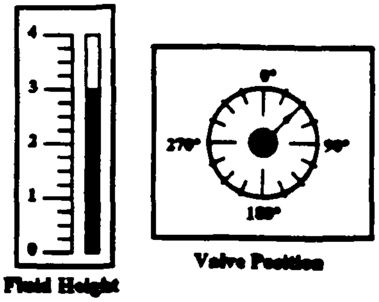


Figure 1.3: Sensors for controlling processes

	0–10°	10–15°	15–20°	20–30°	30–60°	60–180°
0.00–1.50m	1	1	1	1	1	0
1.50–2.50m	1	1	1	1	0	-1
2.50–2.80m	1	1	1	0	-1	-1
2.80–3.00m	1	0	-1	-1	-1	-1
3.00–3.20m	-1	-1	-1	-1	-1	-1
3.20–4.00m	-1	-1	-1	-1	-1	-1

Table 1.1: Table used by the function **table_lookup**

turn_right or **turn_left**. The first turns the valve mounted on the pipe leading into the tank 5° in a clockwise direction, and the second turns the same valve 5° in a counter-clockwise direction. For the time being, we will assume that the changes initiated by these two commands happen nearly instantaneously (*i.e.*, if a **turn_right** command is issued at time t, then $\theta(t + \epsilon) = \theta(t) + 5$, where ϵ is negligible).

Now we can predict future states of the process, but how do we control the process? Perhaps the simplest way is just to experiment and see what works. Suppose that we have done just that, and we have compiled a table that tells us exactly what action to take in every situation. Such a table is shown in Table 1.1. Recall that the task of the controller is to restore the height of the fluid in the tank to the target value of 3 meters. Given information about the current fluid height and valve position, Table 1.1 indicates 1 if the correct action is **turn_right**, -1 if the correct action is **turn_left**, and 0 if the correct action is not to do anything at all. Using this table, we define a simple control algorithm as follows:

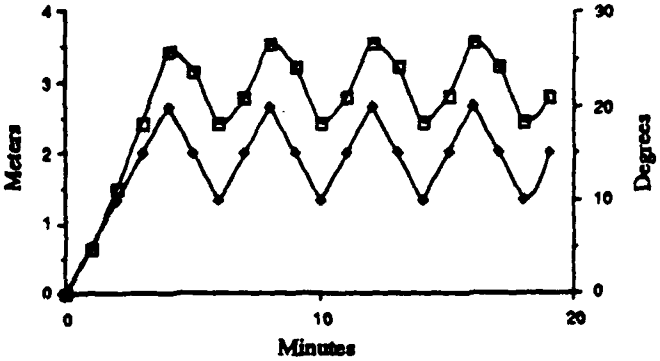


Figure 1.4: The controller's behavior with a 1 minute sample period

```
while true
    wait_for_delay;
    $h$ ← fluid_height;
    $\theta$ ← valve_position;
    $r$ ← table_lookup($h$,$\theta$);
    if $r = 1$
       then turn_right
       else if $r = -1$
               then turn_left
               else do_nothing
```

where fluid_height and valve_position read the corresponding sensors, and table_lookup extracts the appropriate value from the table in Table 1.1 using indices computed from the sensor readings. The procedure wait_for_delay causes the controller to pause for a fixed interval of time referred to as the *sample period*. Figure 1.4 describes the changes in h and θ, with θ controlled by the algorithm described above, the sample period set to 1 minute, and the other variables as set for Figure 1.2.

As an alternative to experimenting in the real world, we could use the model described earlier to experiment with various control strategies for responding to information returned by the sensors. These model-based experiments could then be used to compile a table very much like the one shown in Table 1.1. If the model is reasonably accurate, then the resulting table should look very much like the one developed from experimenting in the real world. Of course, not only do we need an accurate model of the controlled process, but we also need an accurate model of the controller in order to compile an accurate table of responses. So far, we have neglected discussing the controller at all.

In the preceding discussion, we made a number of assumptions (*e.g.*, the valve restricting the output pipe is fixed, and changes initiated by controller commands are nearly instantaneous). Now it is time to review some

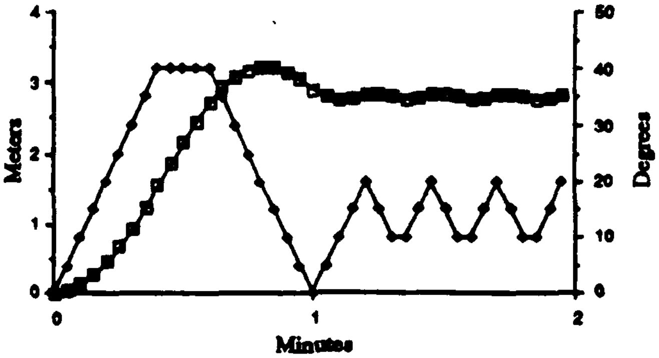


Figure 1.5: The controller's behavior with a 1 second sample period

of those assumptions, and bring to light a numbe r of additional assumptions that were implicit in our discussion of controllers and their response characteristics.

To begin with, we reconsider the role of the sample period in our simple control algorithm. In the description of the algorithm's performance in Figure 1.4, we mentioned that the sample period was set to 1 minute. What if instead we set the sample period to 1 second? Well, for one thing, we would get markedly improved performance, in the sense that the controller would appear to rapidly converge on the target value. Figure 1.5 shows how the controller would respond given a 1 second sample period, assuming that the changes initiated by the commands **turn_right** and **turn_left** occur nearly instantaneously. When we are talking about commands issued every minute, the consequences of such an assumption may be minor, but, if we are talking about commands issued every second, we may be making unrealistic assumptions about the hardware available for carrying such commands. The magnitude of the controller's response is governed by the controller's *gain* (a measure of how fast a controlled variable can change). Generally speaking, the higher the gain, the more massive the controller, the more power it is likely to consume, and the more costly it will be to purchase. Our (implicit) model of the mechanical system for changing the position of the valve is inadequate for a careful analysis of the overall control system.

Another related aspect of the controller's performance that we failed to account for concerns the procedures and how quickly they run on some particular computing hardware. How long does it take to read a sensor? How long does it take to perform all of the auxiliary computations required in the control algorithm? Even table lookup takes time (*e.g.*, time to page the table into memory from disk and compute the indices). Procedures may invoke additional processes whose effects may not be immediately apparent (*e.g.*, the procedure corresponding to **`turn_right`** may take only a few micro seconds to return, but the servo mechanism responsible for actually turning the valve may take several seconds to carry out the command). Suppose that the controller issues the three commands, **`turn_left`**, **`turn_left`**, and **`turn_right`**, in quick succession. Does the second **`turn_left`** command get canceled out by the following **`turn_right`** command, or does the controller swing a full 10° in a clockwise direction before swinging back 5° in a counter-clockwise direction?

Designing good models to capture real-world phenomena can be quite complex. A process model is an abstraction: an idealization appropriate in only a limited context. In the model for a tank filling, we failed to account for evaporation, condensation, malfunctioning valves, other agents adding to or removing from the tank in unpredictable ways, and any of a number of other factors. Correcting for such factors is not simply a matter of providing a more accurate model or better servo mechanisms: hardware has its limitations and, generally speaking, better models take longer to compute and rely upon more detailed information.

Fortunately, lack of precision in the model can be offset somewhat by relying upon the model only for short-term predictions. Feedback through frequent sensing can serve to correct for errors in long term predictions introduced by imperfect or faulty hardware. Sensing and feedback do not, however, obviate the need to take long-term predictions into account. If you expect to be traveling to a foreign country in the next two weeks, you had better check that your passport is in order today; you risk ruining your travel plans by waiting until the last minute.

Another thing to note is that not all predictions are equally useful. It is not necessary—and generally not possible—to predict every consequence of the events that you observe. A little rain may slightly increase the height of fluid in the tank, but the effect is negligible given the flow through the input and output pipes. On the other hand, predicting that someone is about to close the valve mounted on the output pipe could significantly change the optimal strategy for controlling the input valve. As we will see in Chapter [illegible], predicting just those consequences that are useful in guiding behavior turns out to be difficult.

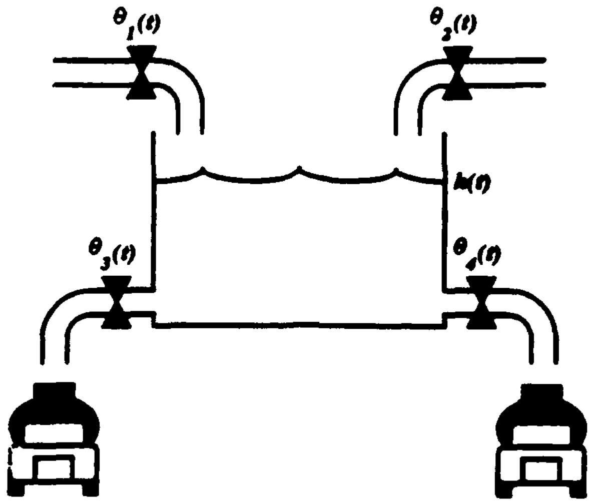


Figure 1.6: A more complex control problem

1.2 Planning

The problem described in Figure 1.1 is rather simple, and it is not difficult to design effective control systems for solving such a problem. Consider what happens when the control problem gets more complicated: several variables to control, other agents to contend with, and some degree of uncertainty about the future. In the situation depicted in Figure 1.6, there are two pipes leading into and two pipes leading out of a tank similar to the one shown in Figure 1.1. Each of the four valves can be manipulated by a separate dedicated servo motor. In Chapter 5, we will consider a variant of this problem in which there is only one servo motor that can be positioned so as to control any one of the four valves. In anticipation of this complication requiring that the controller be mobile, we will refer to the controller as the *robot*.

Now we have to specify what it is that the robot is supposed to do. Figure 1.6 shows a tanker truck positioned under each of the two pipes leading out of the tank. We will assume that at any given time there are zero or more tanker trucks waiting in a queue to be filled up. In addition to controlling the valves on the pipes leading into and out of the tank, the robot can command a truck waiting in the queue to position itself under one of the two pipes leading out of the tank. The two pipes whose valves are labeled θ_1 and θ_2 carry two different chemicals. The control task involves filling each tanker truck with a mixture containing approximately equal proportions of the two chemicals. We will assume that mixing occurs in the tank automatically and instantaneously. Any chemical mixture that flows over the top of the tank is lost and cannot be recovered. The exact proportion of the two chemicals pumped into a tanker truck is not critical, but, if the proportions of the two chemicals in a given truck differ by more than 10%, the contents of the truck will have to be dumped. The robot gets paid for each truck completely filled with an acceptable mixture, and the robot is charged for any chemicals that flow through the two pipes leading into the tank. The robot's task is to maximize its net income.

Maintaining an acceptable mixture is simple if the robot has a separate servo directly controlling θ_1 and θ_2, and the valves have identical flow characteristics; the robot just adjusts the two valves in exactly the same way to guarantee equal proportions of each chemical. The robot can keep the height of the fluid in the tank at any level it chooses, but the higher the level is the faster the mixture will flow through the pipes leading out of the tank, and the faster the robot's earnings will accrue. Of course, there is some risk of spilling fluid if the height is kept too near the top and one of the output valves is suddenly closed, but we will assume that the robot has complete control over all four valves and knows the exact capacity of each truck waiting to be filled.

If we ignore the added task of positioning trucks, the problem of Figure 1.6 is really no more complicated to solve than the problem of controlling a single valve. We could construct a table such as that shown in Table 1.1, or we could derive a fairly simple algorithm to compute the values stored in such a table. Implementing the controller using table lookup is probably not a good idea given the size of the necessary table—the table would have six dimensions (or indices) corresponding to the six state variables: h, θ_1, θ_2, θ_3, θ_4, and the capacity of the next tanker truck waiting in the queue.

Suppose that the robot knows that a tanker truck is within a cubic meter of being completely filled. Using this information, the robot can determine exactly when the valve to the pipe being used to fill the truck should be completely closed. In fact, if there is only one truck to be filled, as soon as the truck is positioned under one of the two pipes leading out of the tank, the controller can use its model of the system of pipes and valves to determine the complete sequence of valve manipulations required to fill the truck as quickly as possible. This idea of using a model to formulate *sequences* of actions is central to planning. In the following, we will examine some of the

```
todo(fill(Truck),Time,ε,
     plan([move(Truck,chute(out1)),
           turn(valve(out1),180°,5°/2min),
           turn(valve(in1),90°,5°/3min),
           turn(valve(in2),90°,5°/3min),
           turn(valve(out1),0°,5°/2min),
           turn(valve(in1),0°,5°/3min),
           turn(valve(in2),0°,5°/3min)],
          [concurrently([2,3,4]),
           concurrently([5,6,7]),
           precedes([1],[2,3,4],0),
           precedes([2,3,4],[5,6,7],capacity(Truck)/5)]))):-
   holds((position(valve(out1),0°),
          position(valve(out2),0°),
          position(valve(in1),0°),
          position(valve(in2),0°),
          in_queue(Truck),length_queue(1)),Time).
```

Figure 1.7: Plan for filling a single truck

advantages and disadvantages of using such a technique. We begin with the advantages.

One can easily imagine a situation in which the robot does not have immediate access to information concerning all of the state variables. For instance, the robot might actually have to do some work to check on the height of fluid in the tank or the position of one of the valves. Rather than constantly perform the work necessary to consult the sensors, the robot can rely upon the model to generate an entire sequence of valve manipulations in advance. We will not discuss how sequences of actions are proposed until Chapter 5; for now, just assume that there is an oracle that produces candidate sequences when asked. The model comes into play when the robot wishes to compare different sequences in choosing the best one. The basic idea is quite simple. Given a sequence of actions, the robot uses the model to simulate the future as it would occur if the actions were carried out. The simulation tells the robot information about how long a particular tanker truck will take to fill and whether or not there is any danger of spilling chemicals using the proposed sequence of actions. This information can then be used to suggest modifications to the proposed sequence of actions. or to compare the proposed sequence with alternative sequences.

It is also possible to simply store an often used sequence of actions. and index it in such a way that it can be easily retrieved when applicable. This is analogous to the method discussed in the previous section for storing responses in tables. For instance. the robot will frequently find itself in the situation where all of the valves are closed. the tank is full. and a truck suddenly appears in the queue. Rather than derive an effective sequence of actions every time it is needed. the robot might store a description of such a sequence of actions—referred to as a *plan*—indexed so that it can be easily retrieved when needed. Figure 1.7 shows a rule for retrieving such a plan. The notation is that of PROLOG. but understanding PROLOG is not necessary for our current discussion.

The rule in Figure 1.7 states that. if all of the valves are closed and there is exactly one `Truck` in the queue at `Time`. then `plan(Steps,Constraints)` is a plan for filling the truck starting at `Time`+ϵ, where the `Steps` consist of seven commands numbered 1–7. and the `Constraints` determine the order in which those commands are to be carried out. Issuing a command of the form `turn(Valve,Angle,Rate)` tells the hardware to turn the `Valve` to the indicated `Angle` (in degrees) at the specified `Rate` (in degrees per minute). A constraint of the form `concurrently(Steps)` specifies that the `Steps` (indicated by their order in the list of plan steps) should begin at the same time and run in parallel. A constraint of the form `precedes(FirstSteps,NextSteps,`Δ`)` specifies that the `FirstSteps` should precede the `NextSteps` with a delay of Δ separating the last step to finish in `FirstSteps` from the first step to begin in `NextSteps`.

If the computations required to derive what to do when a truck suddenly appears in the queue are complex. then having a response stored away for easy retrieval may reduce the amount of time trucks have to wait in the queue. Plans such as the one shown Figure 1.7 can be generated off line and evaluated using a model; complex plans for novel situations can also be constructed on line from simpler plans and evaluated using a model to ensure success. This idea of constructing complex plans from simpler ones is integral to most theories of planning, and we will examine it in greater detail in Chapter 5.

There are also potential disadvantages in generating sequences of actions. The most obvious disadvantage is that the model may be inaccurate. and the sequence of actions will fail to have the desired effect. Unless the controller is really convinced of the accuracy of its model. it will want check that the plan is proceeding according to expectations. This checking is referred to as *monitoring the execution* of a plan. and may involve a considerable amount of effort. If problems are detected. it may be necessary to stop the sequence of actions specified in a plan in order to formulate a new plan or modify steps in the original one. By relying less upon the model, and more upon feedback from sensors, the controller will often save itself a lot of work in generating sequences of steps that are never carried out.

Still, in determining what to do now. it is not as though you can always ignore thinking about what you will do next. Once the controller predicts when a truck will be full. it has to determine what steps are necessary to ensure that the truck's tank does not overflow. It is not enough to say "start closing the valve." Determining when to start closing the valve and how quickly requires anticipating the entire sequence of steps. Keep in mind that a controller only has limited control over its environment: if a valve restricting the flow of fluid into a given truck is wide open. and the truck is nearly full, then the controller will not be able to avoid spilling some fluid. The real issue is not whether or not to plan—planning is an integral part of control—but in what detail to plan. If planning were inexpensive, we would not have to worry about this issue: a controller would always formulate the most detailed plan possible. and there would be no loss if the detailed sequence of steps was not carried out. Unfortunately, planning can be very expensive.

While the problem of Figure 1.6 is a relatively easy one. there are simple modifications that can serve to fundamentally change the problem. Suppose, for instance. that the robot is charged a tax for the time a truck waits between entering the queue and being successfully filled (we will allow the robot to turn away trucks before admitting them to the queue). Now, in addition to its other concerns. the robot has to try to minimize the time trucks spend waiting.

If the robot maximizes the flow of properly mixed chemicals from the tank, and makes sure that full trucks are moved out as quickly as possible and replaced by empty trucks. the only other variable to control is which truck should be filled next. Assuming that the tax is computed as a linear function of the time a truck spends waiting, capacity is the critical factor influencing the choice of next truck. Suppose that the capacity of a truck is an integer-valued quantity. For a given queue of trucks waiting to be filled, the robot will want t assign each truck to one of the two pipes leading out of the tank so as to minimize the amount of time that either one of the two pipes is idle (see Figure 1.8).

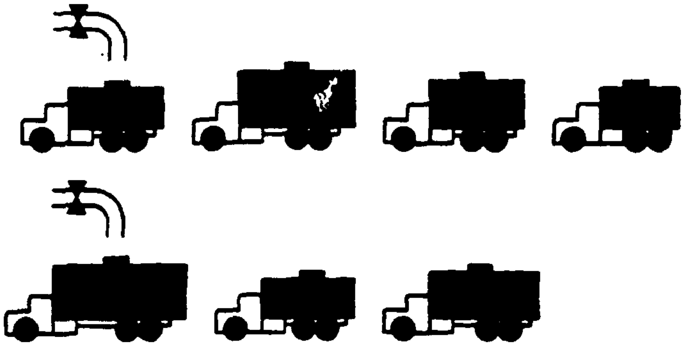


Figure 1.8: Scheduling tanker trucks of varying capacities

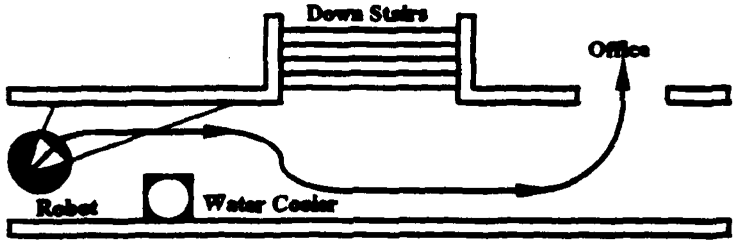


Figure 1.9: A robot navigation problem

Even if we allow that the trucks be instantaneously positioned and the valves instantaneously opened and closed, the problem of assigning the trucks so as to minimize idle time is computationally complex. The problem of determining the optimal assignment of trucks is equivalent to dividing a set of n integers (the capacities of the trucks) into two sets (trucks to be filled from the first pipe and trucks to be filled from the second pipe) so as to minimize absolute value of the difference (time either of the two pipes is idle) of the sum of the integers in the first set (the time the first pipe is being utilized) and the sum of the integers in the second set (the time the second pipe is being utilized). This problem is referred to as the *partition* problem [3], and is known to be in the class of *NP-complete* problems (*i.e.*, the best known algorithms for solving these problems have running times that are at least exponential in the size of their input—the number of trucks in the queue in our case).

For the particular *NP-complete* problem described above, there are good approximate solutions that run in polynomial time. If n is small, it might even be feasible to use an algorithm that computes the exact solution and runs in exponential time. There is a tradeoff involving the time spent in deliberation and the time saved by computing a better answer. While the robot is deliberating about how to fill the trucks, the trucks are waiting in the queue, and the robot is losing money.

It may not seem critical that our robot takes a little extra time in filling the tanker trucks. A simple first-in-first-out strategy for choosing the next truck to fill may prove to be quite effective. There are, however, occasions in which there is more at risk than just a little higher income. Figure 1.9 shows a robot with a single sensor trying to navigate a hallway. In order to avoid hitting the water cooler, the robot has to look to the right; in order to avoid falling down the stairs, the robot has to look to the left. Whether or not the robot can successfully deploy its sensor to avoid both obstacles depends a lot on how fast the robot is moving and how fast the robot can reorient its sensor and interpret the returned data. The designer could take a conservative approach and limit the maximum speed at which the robot can travel so as to ensure the robot's safety, but such a measure is likely to degrade performance significantly. It would be better if the robot could somehow analyze each situation in which it finds itself, weigh its options, and choose the option determined to be best.

The designer of control algorithms has to contend with the inherent limitations of computing hardware and software. There are times when even the simplest algorithms turn out to take too long. For instance, suppose that you wish to track a projectile, and suppose that you have a sensor that returns information concerning the current location of the projectile. By the time you get around to processing the sensor information, it may be out of date, so you will want to label the sensor information with the time that the data was gathered. The obvious thing to do is to label the sensor data using the computer system's on-board clock. The problem is that reading the clock requires loading a procedure into memory, invoking the procedure, and waiting for it to return an answer; all of which takes time, and, more importantly, different amounts of time depending upon how memory is configured, whether or not the procedure has been invoked recently, and any number of other factors. This differential in how long the procedure takes to return an answer can adversely affect the usefulness of the labeled sensor data. For a legged robot trying to walk [6, 2], it can mean the difference between falling or not: for a tennis playing robot [1], it can mean the difference between winning a match or not.

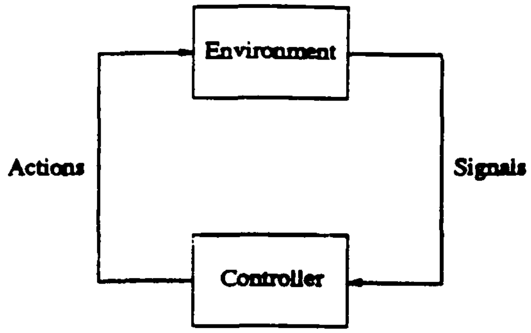


Figure 1.10: A machine coupled to its environment

1.3 Dynamical Systems

Let us return to the question of what it means to control something, and try to answer this question from the perspective of control theory. We begin by providing a general description of a controller coupled to an environment and given some task to achieve.

The controller is represented as a deterministic automaton that takes as input a signal and outputs some action. The environment can be viewed as another automaton that takes as input the controller's action and generates a signal to serve as the controller's next input. The controller is said to be coupled to its environment: the controller and its environment trading blows in a continuous cycle of interaction. Figure 1.10 (after ~~Rosenschein~~ [7]) illustrates this cycle of interaction.

In the following, we describe the interaction between the controller and its environment in terms of a mathematical model called a *dynamical system.* Since we are interested in the behavior of the system over time, we introduce a set of *time points* or *instants*, T. At any given instant. the environment can be in any one of a large number of possible states. This set of states, X, is called the *state space* of the dynamical system.

The controller generally cannot perceive the state of the environment at any given instant, and so we introduce a set of *outputs*. Y, corresponding to what the controller perceives of the state of the environment. Finally, we represent the actions of the controller in terms of a set of *inputs* to the environment. U. Notice that the the terms "input" and "output" assume the perspective of the environment and not the controller; this is a standard convention in control theory, and we adopt it throughout this book.

Unless further qualified (*e.g.*, "the output of the controller"), the terms "input" and "output," refer to, respectively, the input to and output from the environment.

Next, we introduce temporally indexed variables to represent the state, $x(t)$, input, $u(t)$, and output, $y(t)$, at any given point in time, t. We refer to the different ways in which the state, input, and output can evolve over time as *histories*, *time lines*, or, in the parlance of control theory, *trajectories*. The set of all possible state histories or *state-space trajectories* is defined as a set of mappings from time points states.

$$H_X \triangleq \{h_X : T \rightarrow X\}.$$

Similarly, we can define the set of output histories.

$$H_Y \triangleq \{h_Y : T \rightarrow Y\}.$$

We generally restrict the set of state histories by requiring that the evolution of the system state obey certain laws. These laws governing the behavior of the environment are often referred to as the *system state equation(s)*. We represent the state equation by a function that maps states and inputs to states,

$$x(t + 1) = f(x(t), u(t)).$$

Here we employ a difference equation, but we might have used a system of differential equations, a finite-state automaton, a stochastic process, or a set of axioms in a suitable logic. The choice of representation will depend on the structure of time (*e.g.*, integers or the real numbers), the nature of the physical processes we are trying to model, and our own preferences.

Since the controller cannot directly perceive the state of the environment, we also restrict the set of output histories by defining an output function that maps states to outputs corresponding to the signals received by the controller's sensors,

$$y(t) = g(x(t)).$$

This signal invariably contains less information than we would like, and, in most cases, it is noisy and difficult to interpret.[1]

[1] It is the uncertainty resulting from this noisy signal and the fact that information about the state of the environment is frequently delayed in processing that give rise to the need for a systematic treatment of control [5].

So far, we have said nothing about the role of the controller. As with states and outputs, we can define a set of input histories describing the evolution of the actions taken by the controller over time.

$$H_U \triangleq \{h_U : T \to U\}.$$

We restrict input histories according to the hardware and software available to build controllers. We describe the set of possible controllers in terms of functions from the set of sequences of outputs, denoted Y^*, to inputs.

$$P \triangleq \{p : Y^* \to U\}.$$

These functions are called *control laws* or *policies*. In the simplest case, the output function, g, is just the identity function, only the last state is relevant to the decision regarding what action to take, and the set of policies is defined as

$$P \triangleq \{p : X \to U\}.$$

Now we need some objective for the controller to pursue. We begin with a rather ideal objective and define the controller's *task*, K, as a relation on the cross-product space of input/output pairs,

$$K \subset Y \times U.$$

Actually specifying K can be quite difficult given that K indicates exactly what the controller is to do in every possible circumstance.

It may seem more natural to think of a task specified in terms of the best action for a given state,

$$K \subset X \times U.$$

Intuitively, we ought to be able to state the task independent of the particular signals received by the controller. Recall, however, that as far as the controller is concerned, the set of states collapses into a set of equivalence classes determined by the controller's ability to perceive its environment.

Defining a task is a direct method of specifying the desired behavior of a controller. Less direct methods involve somehow specifying restrictions on the state histories of the dynamical system. For instance, we might define a *goal* as a subset of the set of state histories.

$$G \subset H_X.$$

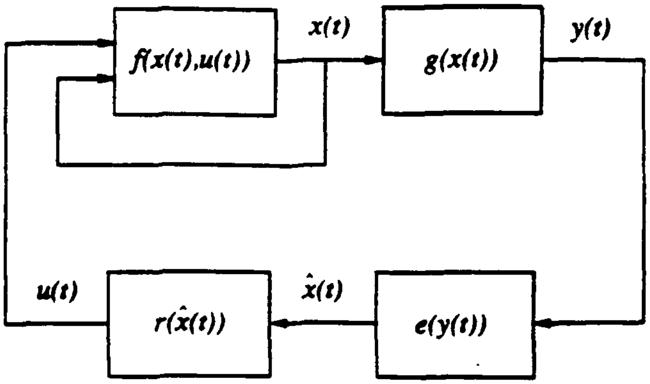


Figure 1.11: A dynamical system

In this case, we wish to find a policy, $p \in P$, such that a controller following p restricts the behavior of the dynamical system to G. Such a policy is said to *achieve* G, and the solution is referred to as a *satisficing* solution.

Alternatively, we might define a *value function*,

$$V : H_X \to \mathbf{R},$$

that allows us to compare different state histories. In this case, we wish to find a policy, $p \in P$, such that a controller following p forces the the state of the dynamical system to evolve according to a history that is maximal with respect to V. Such a policy is said to *maximize* V, and the solution is referred to as an *optimizing* solution. We will refer to the problem of finding a policy to achieve a goal or maximize a value function as the *control problem*.

By providing the controller with a computational model of how certain properties of the environment change over time, we can program the controller to extrapolate from a set of signals to predict what will happen with regard to those properties. A controller equipped with such a model can reason about the consequences of its own actions and those of other processes. It is this aspect of reasoning about change over time that is mostly closely associated with the work in planning. The results of the reasoning are used to construct a *plan* or special-purpose policy to direct the controller's behavior. It is not required, however, that the reasoning be performed by the controller at the time the actions are being executed. The reasoning might be performed at some earlier time and the decisions as to what actions to take compiled into a compact program realizing a particular policy.

As with most complex problems, it is useful to decompose the control problem into component problems. For instance, the control problem is often decomposed into the *state-estimation* or *observation* problem and the *input-regulation problem*. The observation problem is concerned with recovering the system state from the system output. In the simplest case, designing a *state estimator* or *observer* consists of choosing a function from the set,

$$E \triangleq \{e : Y \rightarrow X\}.$$

The output of the observer at time, t, is denoted, $\hat{x}$, indicating that it is an estimate. Similarly, designing a *regulator* consists of choosing a ~~a~~ function from the set,

$$R \triangleq \{r : X \rightarrow U\}.$$

Figure 1.11 shows a block diagram illustrating the various components of a dynamical system and controller.

A good deal of the work in planning implicitly assumes that the observation problem can be solved, and focuses on the input-regulation part of the control problem. But planning need not, indeed should not, be conceived of so narrowly. As we will see, in many problems, the state-estimation and input-regulation problems interact in a complex manner.

There are cases in which we can tackle the control problem by considering the state-estimation and input-regulation problems independently. In the case of linear dynamical systems corrupted by Gaussian noise and subject to quadratic performance criteria, the two problems are said to be *separable*, and the dynamical systems are said to satisfy the *separation property*.

What this means in practice is that one engineer can go off and design an observer that is optimal by some established criterion (*e.g.*, produces an estimate minimizing the expectation of error). Then another engineer can independently design a regulator that is optimal with regard to a second criterion (*e.g.*, optimizes a particular value function over state histories). Separability guarantees that, when the observer and regulator are coupled together, the resulting controller will be optimal with regard to the stated criteria. This means that the actions taken by the regulator have no adverse affect on the ability of the observer to recover the system state. Conversely, the particular measurements taken by the observer have no adverse affect on the ability of the regulator to control the system state.

Note that separability does not hold in general. Consider, for example, what separability would mean for a medical diagnosis and treatment problem. If the problem were separable, then we would not consider the cost of

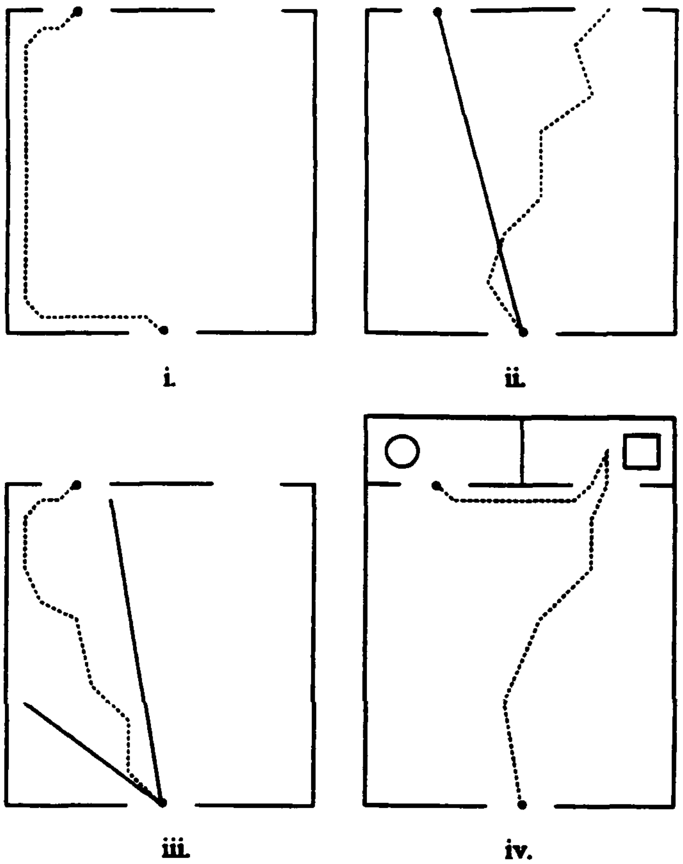

i. ii. iii. iv.

Figure 1.12: Interactions between observation and regulation

performing tests when generating a diagnosis. In particular, there would be no reason to avoid eviscerating the patient in order to determine cause of the symptoms.

As another example, consider the task of a robot navigating in an office environment. Suppose that the robot is required to cross the room shown in Figure 1.12.i. The robot is to enter by the door shown at the bottom of the figure and leave by the door on the ~~right~~ at the top of the figure. Unfortunately, the robot's sensors do not provide accurate information regarding the robot's position and orientation. If the robot remains close to walls and it knows its initial position, it can generally do a good job of keeping track of its position with respect to the room. If, however, the robot roams off into the middle of the room, then it is likely to lose track of its position. In particular, if the robot tries to take the direct path rather than the wall-hugging path as shown in Figure 1.12.ii, then it may very well exit by the wrong door. It is clear in this case that observation and regulation interact strongly.

Planning can play an important role in problems for which the separation property does not hold. By using appropriate models, the controller can reason about the consequences of performing procedures given certain informational states, and, if necessary, design policies that result in the controller obtaining additional information. In Figure 1.12.iii, the controller, possessing a model of the robot's possible movement errors, designs the following plan. While positioned near the door, the controller aims the robot so that by attempting to drive straight it will either go through the door or arrive at a wall at which point it can move to the right hugging the wall to exit by the correct door. This plan is guaranteed to succeed assuming that the controller has an accurate model for movement errors, and will always be better than hugging the wall from very start.

In Figure 1.12.iv, the controller uses a somewhat different strategy. In this case, the controller directs the robot to head straight for the door on the left. The robot exits by the first door it finds, but we assume that the robot can somehow distinguish between the offices that the two doors lead to. If the robot perceives that it is in the wrong office, then it exits the office, and using the wall-hugging strategy to find the office next door.

turns right as it leaves and uses the . . .

The main point of this discussion is that as far as we are concerned the planning problem and the control problem are the same problem. In the rest of the book, we continue to talk about planning and control separately as a means of emphasizing particular issues or techniques closely associated with one or the other of the corresponding academic and engineering disciplines.

1.4 Embedded Systems

The primary computational task of a robot controller is to make decisions concerning what to do next. What to do next is generally thought of in terms of what actuator command to issue next, but there are often other decisions to be made concerning what computations are to be performed and when. Robot decisions are made with regard to certain desirable behaviors (*e.g.*, avoid running into obstacles, or avoid spilling expensive chemicals). These behaviors and the environment in which they are to be achieved de-

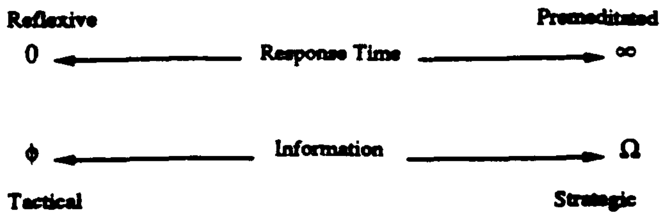


Figure 1.13: Two dimensions of control

termine how the corresponding decision processes are to be implemented. As mentioned in the beginning of this chapter, it is often convenient to distinguish between the controller and the controlled process. We can think about what we would *like* a controller to do, but, when it comes down to building a controller, we have to commit to specific hardware and software, and this commitment will determine what decisions the controller is *capable* of making. The controller is said to be *embedded* in its environment. To analyze a controller, we have to be able to relate the state of the controller and the state of the processes the controller is seeking to control. How well a controller can cope with a given environment will depend upon the amount of time between sensing a situation and being required to respond to that situation, and the availability and volatility of the information potentially useful in deciding how to respond. These factors suggest two dimensions useful in categorizing control problems and their solutions (see Figure 1.13). The less information available and the less time the robot has to process that information, the less likely that the robot's response will account for the possible consequences of its actions. The more information available and the more time that the robot has to reflect on it, the more likely that the robot will be able to generate a response that avoids unpleasant consequences and takes advantage of pleasant ones. These dimensions are quite different from those used to categorize problems and solutions in most areas of computer science.

Computer science concerns itself primarily with *off-line* computing tasks (*i.e.*, ***data processing*** tasks). There are two distinct criteria for such tasks: **correctness** and speed. Most computing tasks in robotics are concerned with ***controlling processes***, and, in particular, controlling processes indirectly and in real time. The notion of correctness in the traditional framework assumes some absolute standard that abstracts away from time. What a control algorithm should compute depends upon the sorts of processes it attempts to control and the information about those processes it can extract from the environment.

Suppose that a controller generates a sequence of actuator commands that *would* have enabled the robot to perform a complex maneuver had they been generated a few seconds earlier. As it is, however, the robot fails to perform the maneuver and tumbles down five flights of stairs. At first blush, it would appear that the controller has failed in its assigned task, but we may be taking too narrow a view. Perhaps the robot was using most of its available computational resources to figure out how to disarm its malfunctioning nuclear self-destruct unit: a task that it did manage to carry out successfully.

talk about
[illegible]

The problem faced by a robot controller is essentially that of optimizing a large number of factors (*e.g.*, time, money, mechanical wear) simultaneously. In order to make such optimizations, a controller has to build up a representation of a complex situation (*e.g.*, one spread out in time and space) and then decide what to do by taking into account how the various pieces of the picture are predicted to interact with one another. For the optimizations to be effective, however, the robot must respond in a timely manner. It would be nice to prove that a given controller satisfied some specified criterion for correct behavior. Unfortunately, for most interesting applications in robotics, such a proof would be prohibitively complex.

Most existing planning systems tend to be far too committed to the plans they formulate and tend to rely heavily on models of the environment and not enough on the environment itself [4]. Such systems do not tailor their decision making to the situation at hand. Given the same abstract task to achieve, these systems will perform the same computations no matter how much time and information is available. They cannot determine when further planning is futile, and they do not have the capability to consider alternative strategies when pressed for time.

Most existing control systems tend to take a rather narrow view of the world and the processes that they seek to control. As long as the world subscribes to the controller's model, these systems behave effectively. Sooner or later, however, unanticipated influences intrude to render the model's predictions inaccurate, resulting in undesirable, and sometimes disastrous, consequences. Building a more complicated model is not always the solution. A complicated model may require more time to compute, thereby reducing the system's response time. An alternative to building a more complicated model is to employ several simple models, each one tuned to a different range of situations. The controller then tries to determine which simple model applies. and changes the model when circumstances dictate. In some sense. this multi-model controller *is* employing a more complicated model. but it is a model that—at least implicitly—takes into account the computational capabilities of the underlying hardware and the anticipated behavior of the processes being controlled. Chapter 8 develops a framework for taking such considerations into account explicitly. in order to dynamically allocate computational resources to suit a given situation.

In subsequent chapters. we will explore a number of methods for constructing and evaluating models of complex systems. We will consider how models are used to control processes. and what sort of tradeoffs have to be made in building effective control systems. The discussion covers both theoretical and practical considerations. The former due to our need to justify design decisions in terms of acceptable mathematical foundations. The latter due to our primary motivation in terms of programming robots to perform useful work. We begin by discussing the theoretical foundations for modeling processes.

Bibliography

[1] Andersson, Russell L., *A Robot Ping-Pong Player: Experiment in Real-Time Intelligent Control*, (MIT Press, Cambridge, Massachusetts, 1988).

[2] Donner, Marc D., *Real-Time Control of Walking*, (Birkhauser, Boston, Massachusetts, 1987).

[3] Garey, Michael R. and Johnson, David S., *Computing and Intractibility: A Guide to the Theory of NP-Completeness*, (W. H. Freeman and Company, New York, 1979).

[4] Georgeff, Michael P., Planning, Traub, J.F., (Ed.), *Annual Review of Computer Science, Volume 2*, (Annual Review Inc, 1987).

[5] Koditschek, D., Robot Control Systems, Shapiro, Stuart, (Ed.), *Encyclopedia of Artificial Intelligence*, (John Wiley and Sons, New York, 1987), 902–923.

[6] Raibert, Marc H., *Legged Robots That Balance*, (MIT Press, Cambridge, Massachusetts, 1986).

[7] Rosenschein, Stan, *Formal Theories of Knowledge in AI and Robotics*, Technical Report CSLI-87-84, Center for the Study of Language and Information, 1987.

Chapter 2

Dynamical Systems

For our purposes, a *process model* is a device that, given certain information about the state of a physical system, enables us to determine certain other information about that system. The device usually includes some mathematical characterization of the system's properties and how they relate to one another. It also includes some sort of a calculus whereby an engineer or a machine can compute the predictions of the model given some initial conditions.

Process models are used by engineers to design control systems. In some cases, the process model is used only to evaluate a given controller. In other cases, the process model becomes an integral part of the control system. In this chapter, we consider a few of the large number of process modeling techniques available to the engineer, and develop some notation for describing process models that will be used in subsequent chapters.

2.1 Constructing Physical Models

To construct a model for a process, we have to identify those properties of the world that determine the behavior of the process. First, there are those properties that prompted our interest in the process to begin with. In the case of the tank-filling process described in Chapter 1, we are primarily interested in the height of the fluid in the tank. Second, there are those properties that affect the properties that we are interested in. In order to account for the level of fluid in the tank, we have to know the dimensions of the tank, the flow characteristics of the input and output pipes, and the

[0]©1990 Thomas Dean. All rights reserved.

position of the valves. It is easy to underestimate the difficulty of this part of the modeling task.

Textbooks typically just give the student the set of physical properties that he or she needs to be concerned with. There is an implicit assumption that these are all and only the properties that need to be considered. How do we know that the temperature of the fluid does not affect the height of the fluid in the tank? Well, of course, we don't know this. The temperature may affect the fluid height by changing the rate at which the fluid evaporates; however, given that the temperature does not vary substantially, the effect of temperature on fluid height is negligible.

Almost any property of the world *can* have an impact on the level of the fluid in the tank: agricultural trends affect global weather patterns that affect local temperature and humidity that ultimately affect fluid height. The predictions made by a particular model are likely to be accurate only if certain assumptions hold. Whether or not to account for a given property of the world in a particular model depends on a number of factors: the magnitude of the effect (*i.e.*, does it result in substantial changes in the properties of interest), the probability of the effect (*i.e.*, do the changes occur with high frequency), and the complexity of the model (*i.e.*, what additional computations are required to account for the property in the model).

This last is particularly important, and, yet, it is often overlooked in evaluating a model. There is often some utility in getting an answer to a question quickly. If this were not the case, you would always want the model that makes the most accurate predictions possible. Given that time has to be taken into account, there is a tradeoff to be made regarding the accuracy of the model and the time that it takes to compute its predictions.

The following sections describe some basic methods for modeling physical processes in control theory. Section 2.2 considers the use of the differential and integral calculus for modeling processes and analyzing the behavior of control systems, focussing on ideas from classical control theory. Section 2.3 considers the general problem of modeling dynamical systems and introduces ideas from linear system theory, drawing upon results from modern control theory.

2.2 Mathematical Modeling in Control Theory

Much of control theory depends on the use of mathematical models based on the techniques of the integral and differential calculus. These techniques enable the control theorist to model a wide variety of mechanical. electrical. fluid. and thermodynamic systems. By modeling both the controlling process and the process being controlled as a set of differential equations, the control theorist is able to analyze behavior of the combined system. and predict the performance characteristics of the controlling process (*e.g.*. how fast the system responds to a disturbance or change in input). In this section, we summarize some of the issues involved in modeling physical systems using the techniques of control theory.

Anyone who has taken a course in differential equations or advanced calculus has seen numerous examples of mathematical models of physical systems. Most introductory texts on the differential calculus include idealized models of population growth. the decay of radioactive materials. and the fluctuation in prices as a function of supply and demand. If you took a physics course. you were early on exposed to Newton's laws of motion. Newton's second law of motion states that the product of a body's mass and the acceleration of its center of mass is proportional to the force acting on the body. Let x be a function that depends on t and denotes the position of the center of mass of the object as measured from some fixed point along a vertical line. Let M be the mass of the object. and $\mathcal{F}$ be the force acting on the object in the direction of travel. The following differential equation

$$M\frac{d^2x}{dt^2} = \mathcal{F} \qquad (2.1)$$

is called the *equation of motion* of the body.[1] If we know something about the forces acting on the body. then we can use this equation to make predictions about the motion of the body.

If x is the directed distance upward of the object as measured from the surface of the earth. and v_0 is the object's initial velocity. then. assuming that the only force acting on the object is gravity. Equation 2.1 becomes

$$M\frac{d^2x}{dt^2} = -Mg \qquad (2.2)$$

[1] To simplify the discussion. we implicitly adopt the standard system of units for measuring mass. distance. and time so that the constant of proportionality is one.

where is g is the acceleration due to gravity near the surface of the earth. We can solve this simple second-order differential equation. by integrating twice and using the initial conditions to determine the constants of integration. The following formula

$$x(t) = -\frac{1}{2}gt^2 + v_0t \tag{2.3}$$

describes the position of the object at $t \geq 0$ given the initial conditions

$$x(0) = 0. \qquad \frac{dx(0)}{dt} = v_0,$$

and assuming that the object is propelled upward at time $t = 0$. From Equation 2.3, we can predict the maximum height ($v_0^2/2g$) reached by the object and the time it takes the object to fall back to the surface of the earth ($2v_o/g$). Equation 2.3 together with tools of the differential calculus provide us with a simple model of an object falling through a gravitational field.

We know that Equation 2.3 is only approximate in that it neglects several important influences on objects falling through a relatively dense atmosphere under the influence of gravity. For instance, Equation 2.3 treats gravity as a constant acceleration whereas we know that Newton's inverse square law provides a more accurate estimate of the force due to gravity acting on an object. If the earth is assumed to be a sphere of radius R. and r denotes the distance from the center of mass of the object to the center of the sphere. then

$$M\frac{d^2x}{dt^2} = -\frac{MgR^2}{r^2}$$

can provide a more accurate estimate of the position of the object than that provided by Equation 2.1. especially in the case of an object that travels a significant fraction of the distance R.

We can also account for the damping force exerted on the object by the atmosphere as the object moves along its trajectory. If the damping force is proportional to the object's velocity, and C is the damping constant. then

$$M\frac{d^2x}{dt^2} = -\frac{MgR^2}{r^2} - C\frac{dx}{dt} \tag{2.4}$$

will, at least potentially, provide a better estimate than equations that neglect friction. Potentially, because, having identified that some property of the environment influences a particular process, you still have to determine the form and the magnitude of that influence. There are situations in which

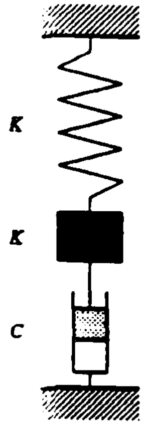


Figure 2.1: A spring-mass-dashpot system

the damping force is more nearly proportional to the square or the cube of the velocity. In addition, the damping "constant" may not be constant at all, dependent as it is on the shape of the object and the density of the air through which the object is moving. If you are not careful, you can actually reduce the predictive accuracy of a model by trying to account for additional properties.

As another example of physical modeling, Figure 2.1 shows a block of mass M suspended from the ceiling by a spring and connected by a rigid rod at its base to a damping device called a *dashpot*. The spring counteracts the force of gravity and the dashpot tends to inhibit vertical motion in either direction. Suppose that the force exerted by the spring is equal to the product of the distance that the spring is stretched or compressed and K, the spring constant. Let d be the distance past the spring's resting length such that the force of the spring completely offsets the force of gravity, and the block will remain at rest (*i.e.*, $Mg = Kd$). The equation of motion for the block, neglecting the dashpot, is

$$M\frac{d^2x}{dt^2} = Mg - K(x + d) = Kx. \qquad (2.5)$$

To account for the dashpot, we assume that the damping action of the dashpot is proportional to the velocity of the block and introduce another term into Equation 2.5. The result is

$$M\frac{d^2x}{dt^2} + C\frac{dx}{dt} + Kx = 0 \qquad (2.6)$$

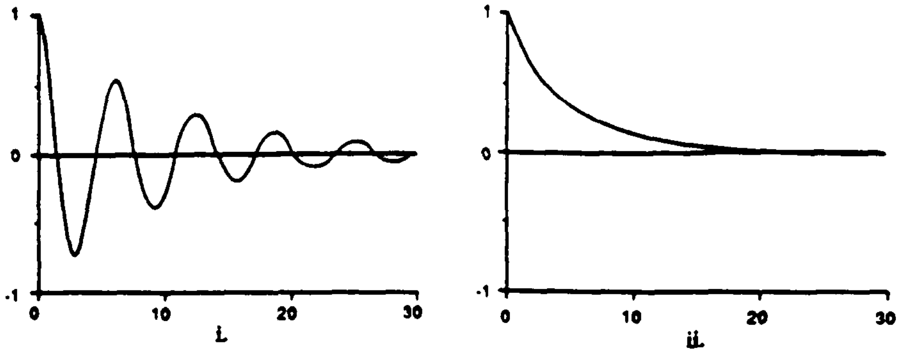


Figure 2.2: Response of the spring-mass-dashpot system in the (i) underdamped and (ii) overdamped cases.

where C is the damping constant.

There are three different solutions to Equation 2.6 depending on whether the quantity C^2 is less than, greater than, or equal to the quantity $4MK$. These solutions correspond to the underdamped, overdamped, or critically damped cases. If $C^2 < 4KM$, then the specific solution to Equation 2.6 that satisfies the initial conditions,

$$x(0) = x_0, \qquad \frac{dx(0)}{dt} = 0,$$

is given by

$$x(t) = x_0 e^{\alpha t}\left(\cos \omega t + \frac{\alpha}{\omega} \sin \omega t\right),$$

where

$$\alpha = \frac{C}{2M}, \qquad \omega = \frac{1}{2M}(4MK - C^2)^{1/2}.$$

In this (the underdamped) case, the mass oscillates about the equilibrium point, its amplitude decreasing exponentially with time as shown in Figure 2.2.i. If $C^2 > 4KM$, then the specific solution to Equation 2.6 satisfying the same initial conditions is given by

$$x(t) = \frac{x_0}{\beta - \alpha}\left(\beta e^{-\alpha t} - \alpha e^{-\beta t}\right),$$

where

$$\alpha = -\frac{1}{2M}\left[-C + (C^2 - 4MK)^{1/2}\right], \quad \beta = -\frac{1}{2M}\left[-C - (C^2 - 4MK)^{1/2}\right].$$

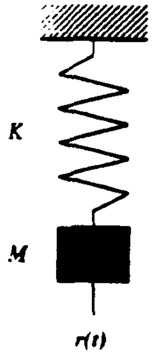


Figure 2.3: An external force acting on a spring-mass system

Figure 2.2.ii illustrates the behavior of the resulting overdamped system. The important thing to note here is that. assuming M is fixed. we can vary K and C to achieve different behaviors.

Control theorists are often interested in how a physical system responds to a particular input signal. The *step input*. corresponding to a fixed-size instantaneous change in the reference or a disturbance, provides a convenient basis for comparing performance. In the case of the spring-mass-dashpot, a step input might correspond to the block being displaced from its equilibrium point or given some initial velocity. Equation 2.6 might serve as a simple model for an automobile shock absorber. The input signal would correspond to a force acting on the mass (*e.g.*. the automobile hitting a bump in the road). The engineer designing such a system is interested in the characteristics of the output signal corresponding to the changes in the position of the mass. In particular. the engineer wants to know whether or not the control system he or she designs is *stable*. A system is said to be stable if its response to a bounded input is itself bounded. In the case of our spring-mass-dashpot system. if we displace the mass a small amount from its equilibrium point, it will eventually return to that point. Similarly. if we give the mass some small initial velocity, it will also eventually return to its equilibrium point.

Unstable systems can manifest undesirable and sometimes violent behavior (*e.g.*, thermal runaway in a nuclear power plant). Suppose that we eliminate the dashpot from our spring-mass-dashpot system and introduce an additional, external force acting on the mass as pictured in Figure 2.3.

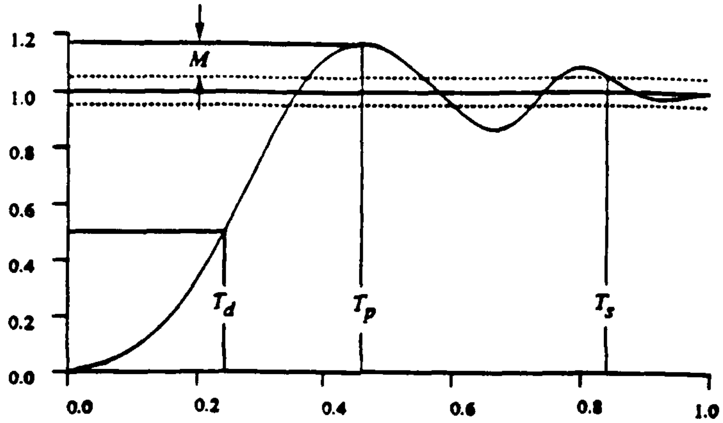


Figure 2.4: Transient response to a step input indicating T_d (*delay time*) the time required for the controlled variable to reach 50% of the target, T_s (*settling time*) the time required for the controlled variable to achieve and maintain a value ±5% of the target. T_p (*peak time*) the time at which the controlled variable achieves the largest value above the target, and M (*peak overshoot*) the largest value of the controlled variable above the target.

Suppose that the external force is periodic of the form

$$r(t) = R\sin\omega t$$

where R is a positive constant. The equation of motion is

$$M\frac{d^2x}{dt^2} + Kx = R\sin\omega t.$$

If $\omega = (K/M)^{1/2}$, then the amplitude of the oscillations will increase due to the phenomenon of resonance [10]. The model predicts that the oscillations will increase indefinitely, but, of course, there will come a point past which the mathematical model is no longer appropriate and other physical properties will come into play (*e.g.*, the spring breaks or the device generating $r(t)$ reaches saturation).

Stability is just one aspect of a system's *transient response* to a step input (*i.e.*, the behavior of the system in transition from one stable state to another as a result of a step input). An engineer usually is also interested in the system's *settling time* (*i.e.*, the amount of time it takes the system to achieve a state in which .ie value of the controlled variable is within some small percentage of the target value). the system's *steady-state error* (*i.e.*. the percent error of the system in the limit). and the system's *overshoot* (*i.e.*. the maximum past the target that the system achieves in responding to step input). Figure 2.4 illustrates some of the important characteristics of a system's transient response to step input [6. 12].

Peak overshoot is a particularly important transient response characteristic in a number of applications. In some cases, the sort of underdamped behavior shown in Figure 2.2.i is unacceptable. In attempting to restore equilibrium. the system overshoots the target or equilibrium point. In the case of a robot arm positioning a part. overshoot might correspond to the part striking a surface. In the case of the liquid-level system of Chapter 1. overshoot might mean that the level of fluid in the tank goes above the top of the tank, spilling fluid on the floor.

A good deal of control theory is concerned with analyzing the performance of control systems with regard to criteria such as stability, settling time. steady-state error, and overshoot. One way to analyze a control system is to build a mathematical model as a system of differential equations. solve the equations. and then examine the behavior of the system in the time domain. This is essentially what was done in our analysis of the spring-mass-dashpot system above. This method of analysis can be complicated by the fact that the equations for any reasonably complex control system are likely to be difficult to solve. and. in order to find parameters for the control system that provide good performance, it may be necessary to to look at a large number of special cases. While there exist effective methods for analyzing control systems in the time domain. one of the great successes of what is called *classical control theory* has been the development of mathematical techniques that enable an engineer to recast a control problem as a problem in the frequency domain. Most of these techniques rely of the use of the *Laplace transform*.

The Laplace transform enables the control theorist to avoid working with differential equations by replacing these generally difficult-to-solve equations with simpler algebraic equations. Since the Laplace transform exists for many linear differential equations encountered in control systems design, methods based upon the use of the Laplace transform are widely employed in the analysis of control systems. The Laplace transform of a function of time. f(t). is defined as

$$F(s) = \int_0^\infty f(t)\epsilon^{-st}dt = \mathcal{L}(f(t)). \qquad (2.7)$$

The Laplace transform of the derivative of a function can be obtained from Equation 2.7 using integration by parts

$$\mathcal{L}\left(\frac{df(t)}{dt}\right) = s\mathcal{L}(f(t)) - f(0).$$

However. it is usually not necessary to derive the Laplace transform of a function every time that the engineer is faced with a new problem. Tables of functions and their Laplace transforms have been compiled for most functions commonly encountered in engineering applications.

The Laplace transform of a sum of two functions is just the sum of the Laplace transform of the first function with that of the second. Using this fact and the tables of Laplace transforms, the control engineer can rather easily obtain the Laplace transform for many differential equations used in modeling physical systems. The advantage is that the resulting algebraic equation usually can be easily solved for the variables of interest. The *transfer function* of a control system is defined to be the ratio of the Laplace transform of the input variable to the Laplace transform of the output variable. By analyzing a control system in terms of the relation of the Laplace transform of the inputs to the Laplace transform of the outputs. it is possible to gain a good understanding of the system's performance properties.[2]

To make the analysis of control systems even easier. there are tables that provide the transfer functions for many of the differential equation relations encounted in control systems. An engineer can design a control system using various control components connected to one another by the way in which they pass signals. From these separate components, the engineer can derive the transfer function for the complete control system algebraically. The familiar block diagrams displayed in the control theory literature provide a convenient graphical representation of the underlying process model. The

[2] Frequency-domain methods involving transfer functions are so named because they allow the engineer to analyze the behavior of a system in terms of its response to inputs of varying frequencies and amplitudes. By evaluating the transfer function. $T(s)$. at $s = j\omega$ for any $\omega \in \mathbf{R}^+$. we obtain a complex number. $T(j\omega) = \alpha(\omega) + j\beta(\omega)$. whose magnitude. $\sqrt{\alpha^2(\omega) + \beta^2(\omega)}$, represents the response of the system in steady state to a sinusoidal input of frequency. ω. in terms of the ratio of the output to the input amplitude.

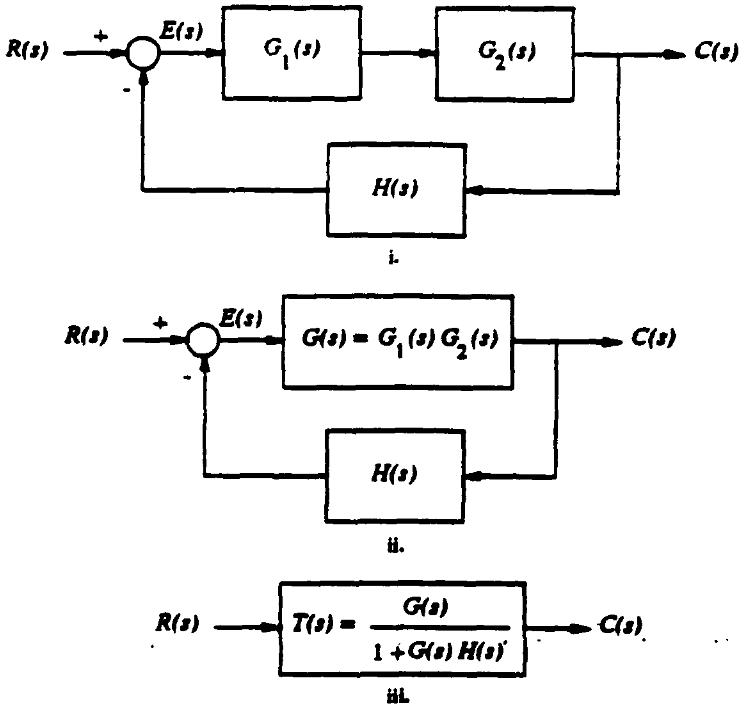


Figure 2.5: Block diagram of a control system utilizing feedback

boxes in such diagrams are usually labeled with the transfer function for the corresponding system component and the arcs indicate the signals passing between components. Figure 2.5.i depicts the block diagram for a control system in which the output of the plant is fed back through some sort of a filter or amplifier and combined with the input to provide an error signal used by a compensator in controlling the plant. The control system pictured in Figure 2.5.i illustrates a simple instance of error-driven feedback. in which the system reference signal is continuously compared with the system's output in order to adjust various system parameters.[3]

Block diagrams can be simplified by algebraically combining the transfer functions of connected components according to a few simple rules [6]. For instance. the two blocks labeled $G_1(s)$ and $G_2(s)$ in Figure 2.5.i can be combined to form.

$$G(s) = \frac{C(s)}{E(s)} = G_1(s)G_2(s).$$

noting that $C(s) = E(s)G_1(s)G_2(s)$. The simplified block diagram is shown in Figure 2.5.ii. The simplest block diagram is just a single box labeled with the transfer function for the complete control system. For instance, we can reduce the block diagram for the system shown in Figure 2.5.ii to a single component with input $R(s)$, output $C(s)$, and transfer function.

$$T(s) = \frac{C(s)}{R(s)} = \frac{G(s)}{1 + G(s)H(s)},$$

noting that $E(s) = R(s) - H(s)C(s)$ and $C(s) = E(s)G(s)$. This simplest block diagram is shown in Figure 2.5.iii. The function. $T(s)$, known as the *closed-loop transfer function*. is the basis of many existing control systems.

Much of the control theory found in textbooks deals with what are called *linear systems*. A system is said to be linear in terms of inputs and outputs if and only if it satisfies the properties of *superposition* and *homogeneity* [6]. A system satisfies the property of homogeneity if for any constant K and input x for which the output of the system is y, if the system is input Kx. the system outputs Ky. A system satisfies the superposition property if for any two inputs x_1 and x_2 with corresponding outputs y_1 and y_2, if the system is input $x_1 + x_2$, the system outputs $y_1 + y_2$. At first blush. the restriction to linear systems would seem to relegate much of control theory to a purely academic pursuit given that most natural systems are nonlinear at least

[3] In some texts, error-driven feedback is synonymous with *unity feedback*. corresponding to the case in which $H(s)$. in Figure 2.5.i. is the identity function.

in some range of their variables. Fortunately, we can develop reasonably accurate linear approximations by identifying almost-linear regions in the operating range of nonlinear systems. If the natural operating conditions of a system vary over a wide range, it may be necessary to develop several linear approximations and switch between them when necessary. This method of switching between controllers is the basis for a technique used in adaptive control called *gain scheduling.*

Other approximations are often made to simplify analysis and implementation. For instance, it is often possible to eliminate some of the higher-order terms in a model involving differential equations. By eliminating the higher-order terms, the subsequent analysis may ignore effects due to high-frequency inputs. Hopefully, these effects will not pose a problem in practice, but no model should be relied upon without careful experimentation comparing the performance of the modeled system with that of the real one.

While we have emphasized modeling continuous processes, control theory provides tools for modeling discrete processes as well. The discrete analog of a differential equation is called a *difference equation* and is used extensively not only to model discrete systems, but also to approximate continuous systems using digital hardware. Analog computers still play an important role in engineering, but, with the introduction of inexpensive digital computing hardware, a great deal of attention has been given to discrete modeling techniques.

Digital computers are limited in that they can only sample system variables at discrete points in time. Usually, the delay between samples is fixed of duration τ. By introducing a new complex variable

$$z = e^{s\tau},$$

we can define a discrete version of the Laplace transform called the z-transform for a discrete function $f(k)$ as

$$\mathcal{Z}(f(k)) = F(z) = \sum_{k=0}^{\infty} f(k) z^{-k}.$$

There exist techniques, analogous to those based on the Laplace transform, for using the z-transform to analyze the response characteristics of control systems [3]. Analysis using the z-transform is complicated somewhat by the fact that information is irretrievably lost in a sampled system. It is generally necessary to identify the various frequency components of the

Nyquist

input signal in the Fourier domain, and adjust the sampling rate accordingly to avoid effects due to signai aliasing (*i.e.*, mistakenly associating high frequency components of ~~the signal~~ with lower frequency components). According to a theorem of ~~Claude~~ Shannon aliasing can be avoided entirely by ensuring that the sampling frequency ($1/\tau$ samples per unit time) is at least twice the frequency of the highest frequency component of the input signal [4]. Of course, it may not be possible for the digital hardware to sample that quickly or perform the necessary computations required to generate an appropriate response. The problem of implementing complex control strategies that keep pace with a rapidly changing environment will be addressed frequently in this monograph.

There exist processes for which we know the form of an appropriate model (*e.g.*, we know that the process can be modeled using a kth-order linear differential equation with constant coefficients), but we do not know the parameters of the model. For instance, the system we are trying to model might be a black box that we know to be a single-input single-output linear system, but the model parameters do not correspond to any known physical parameters such as the spring constant or the damping constant in the model for the spring-mass-dashpot system. In this case, it may be possible to find values for the parameters of the model by sampling the input and output of the system, and "fitting" the parameters of the model to the data. This is a special case of what is called *system identification*, and constitutes an important part of the branch of control theory known as *adaptive control* [1, 11]. System identification can be done off line during the design of the control system as prologue to the sort of analysis described above. In adaptive control, system identification is done on line by the control system, and the results of system identification are used to adjust the parameters of a controller. This approach to control is particularly useful if the physical system that you are attempting to model changes over time (*e.g.*, a plant with mechanical parts that are subject to wear).

One particularly convenient feature of the mathematical models used in control theory is that, at least as far as the analysis is concerned, what one learns about design in one area is immediately applicable in another area for which there exists appropriate analogical apparatus mapping the variables between the two systems [6]. For instance, the engineer familiar with the analysis and design of electrical control systems can often apply what he or she knows to the analysis and design of mechanical or fluid control systems. The basic models and their corresponding equations appear again and again, and hence much of what is learned can be compiled into tables, tools, and cookbook-style methods for dealing with commonly occurring spec ic cases [4].

In this section, we considered some of the basic techniques involved in modeling physical systems. We briefly touched upon some of the methods and terminology of control theory, specifically what is referred to as classical control theory. As was mentioned, classical control is most closely associated with analysis in the frequency domain. In the next section, we introduce a particular class of physical systems important from the standpoint of control, and consider modeling techniques drawn from modern control theory.

2.3 Modeling Dynamical Systems

The techniques described in the previous section are primarily useful for physical systems that can be modeled with a single input and a single output variable. In this section, we consider systems modeled with any finite number of input and output variables. We restrict our attention to a limited class of physical systems called *dynamical systems*. A dynamical system is defined by the following mathematical objects and axioms governing them.[4]

- A set of time points $T \subset \mathbf{R}$
- A set of states X
- A set of inputs U
- A set of outputs Y
- A set of input functions

  $$\Sigma = \{\sigma : T \to U\}$$

why T ?
ANSWER
time VARIANT syst
next page

- A state transition function

  $$f : T \times T \times X \times U \to X$$

  whose value is the state $x(t) = f(t; \tau, x, \sigma) \in X$ resulting at time $t \in T$ starting from an initial state $x(\tau)$ at time $\tau \in T$ influenced by the action of the input σ.

[4] The definitions provided here roughly follow those of Kalman [9] though we have sacrificed rigour in some places to avoid lengthy technical commentary. Our objective here is to set the stage for a discussion of practical methods, and not, as in the case of Kalman's work, the precise description of mathematical abstractions.

√ T ?

- An output function
$$g : T \times X \to Y$$

We impose some additional restrictions. In particular, for any $t_1 < t_2 < t_3$ and $\sigma \in \Sigma$ we have

$$f(t_3; t_1, x, \sigma) = f(t_3; t_2, f(t_2; t_1, x, \sigma), \sigma),$$

and for any two input functions σ and σ' that agree on the interval $\langle t, \tau \rangle$ we have

$$f(t; \tau, x, \sigma) = f(t; \tau, x, \sigma').$$

The first of these restrictions provides a reasonable property that allows us to compose inputs. The second is often referred to as the *principle of causality* [2].[5] Given an input function $\sigma \in \Sigma$ and an interval of time $(t_1, t_2]$, an *input segment* $\sigma_{(t_1, t_2]}$ is just σ restricted to $(t_1, t_2]$. We require that, if $\sigma, \sigma' \in \Sigma$ and $t_1 < t_2 < t_3$, then there exists $\sigma'' \in \Sigma$ such that $\sigma''_{(t_1, t_2]} = \sigma_{(t_1, t_2]}$ and $\sigma''_{(t_2, t_3]} = \sigma_{(t_2, t_3]}$. This last property is called *concatenation of inputs* [9], and provides us with a useful closure property for the set of input functions.

We also assume that the response of a dynamical system is independent of the particular time at which it is exercised. We say that a dynamical system is *time invariant* if the following properties hold.

- T is closed under addition.
- Σ is closed under the *shift operator*, $z^s : \sigma \mapsto \sigma'$, defined by
$$\sigma'(t) = \sigma(t + s)$$
for all $s, t \in T$.
- For any $s, t, \tau \in T$, we have
$$f(t; \tau, x, \sigma) = f(t + s; \tau + s, x, z^s \sigma)$$
- The output function $g(t, .)$ is independent of t.

[5] There is a tendency in mathematical control theory to refer to certain assumptions or restrictions as principles. This is particularly the case where the mathematics would be difficult or impossible without imposing some restrictions. In some cases, such as the principle of causality described here, the restrictions seem innocuous enough, but in others they appear to motivated by nothing more than mathematical convenience or necessity. Witness the fact that superposition, which underlies linearity, is often introduced as the "principle of superposition" [9].

We will be concerned with *continuous time* dynamical systems (*i.e.*, T is the real numbers) and *discrete time* dynamical systems (*i.e.*, T is the integers). For mathematical purposes, we may introduce additional restrictions such as smoothness and linearity, but it should be pointed out that many physical systems cannot be modeled exactly under such restrictions.

We represent a continuous time-invariant dynamical system as

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= g(x(t), u(t)) \end{aligned}$$

where the first equation is called the *state equation* and the second the *output equation*. The state and output equations typically consist of differential equations such that for any initial state $x(t_0)$ and input u both equations have unique solutions. The discrete counterpart of the continuous system is represented as

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) \\ y(k) &= g(x(k), u(k)) \end{aligned}$$

where the state equation in this case is a difference equation.

So far, we have treated states, inputs, and outputs as simple unstructured sets. Generally, the states, inputs, and outputs have considerable structure; it is often reasonable to represent each in terms of a multidimensional vector space (*e.g.*, $\mathbf{R}^n$). Each dimension of the space corresponds to a component *variable* of the corresponding vector space. For instance, in designing a dynamical system to model the fluid flow in and out of a holding tank, we might employ three state variables, the height of the fluid in the tank, the angle of the input valve, and the angle of the output valve. The resulting state space would be a subset of $\mathbf{R}^3$. In designing a system to model a robot, we might use the position in x, y, and z, and orientation in $\theta_{x,y}$, $\theta_{y,z}$, and $\theta_{x,z}$ for a six-dimensional state space, $\mathbf{R}^6$. In general, the state, input, or output variables may be boolean, real, integer, or discrete valued, and can correspond to any representable quantity or its derivatives, as long as the resulting space satisfies the requirements for being a finite-dimensional vector space [5]. By characterizing the states, inputs, and outputs in terms of linear vector spaces, we can bring to bear the considerable power of linear algebra and linear systems theory.

Much of linear control is concerned with linear time-invariant systems of the form

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$$

$$y(t) = Cx(t)$$

where $\mathbf{x}$ is the n-dimensional state vector, $\mathbf{u}$ is the p-dimensional input vector, $\mathbf{y}$ is the q-dimensional output vector, and A, B, and C are, respectively, $n \times n$, $n \times p$, and $q \times n$ real constant matrices.

As a simple example illustrating how to construct a linear dynamical system, consider a single-degree-of-freedom robot of mass, M, acted upon by a force, $\mathcal{F}$. Let z be the position of the robot in some arbitrary frame of reference. We assume that the plane of motion is horizontal and that there are no frictional forces acting on the robot. The relationship between position, z, and the force, $\mathcal{F}$, is completely determined by Newton's second law of motion.

$$M\ddot{z} = \mathcal{F}$$

The dynamic behavior of the robot can be described in terms of the position and velocity of the robot, and, hence, we define the state vector to be,

$$\mathbf{x}(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix}.$$

Eqnating the system ontput and the system state, we can write down the state and output equations as follows.

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 1/M \end{bmatrix} \mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{x}(t) \end{aligned}$$

defined

Generally, the system output contains incomplete information from which it is necessary to reconstruct the system state. In subsequent chapters, we consider some of the issues involved in attempting to infer the system state from incomplete information.

The restriction of linearity is a critical one that causes some researchers to dismiss much of mathematical control theory as a purely academic pursuit with no practical consequences. Most physical systems are nonlinear, and, hence, we can only approximate these systems using linear models. In many cases, such approximations are valid over only a limited range of the systems operating conditions. While these problems make it difficult to apply results from linear systems theory, the methods of linear systems theory are so powerful that the effort is often well spent. It seems unlikely that a general method for analyzing nonlinear systems will emerge [7], and that instead

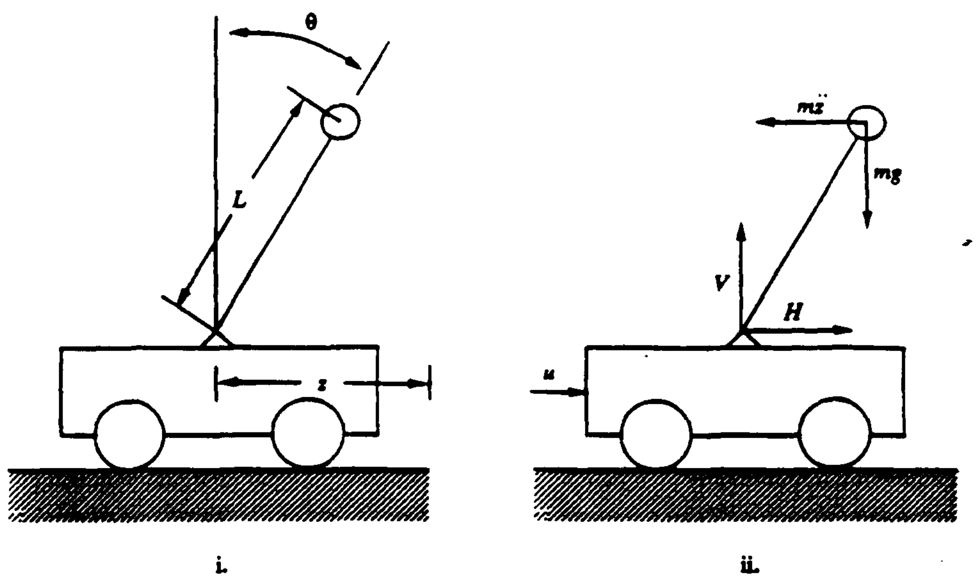


Figure 2.6: Inverted pendulum mounted on a cart

researchers will divide the class of nonlinear systems into a set of more manageable subclasses for which there exist special methods of analysis, much of which will be based on ideas drawn from linear systems theory.

To illustrate how to approximate a nonlinear system by a linear one, we consider a classic example in control that involves modeling an inverted pendulum mounted on a cart that can move back and forth along a horizontal track. This problem is often cited as an analogue of the problem of controlling a missile balanced atop its booster rockets [6, 8]. The presentation here follows that of Gopal [8]. We assume that the controller can exert a force on the cart to propel it to the right or left along the horizontal track. Let z be the horizontal position of the cart's center of gravity, and $z + L\sin\theta$ the horizontal position of the center of gravity of the pendulum, where L is the distance from the pivot to the center of gravity of the pendulum. Similarly, $L\cos\theta$ is the vertical position of the center of gravity of the pendulum. Figure 2.6.i shows the basic configuration of cart and pendulum.

The state of the system is completely described by the position and velocity of the cart and the angular position and angular velocity of the pendulum. Thus we have the state vector:

$$\mathbf{x}(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \\ \theta(t) \\ \dot{\theta}(t) \end{bmatrix}.$$

In order to set up the dynamical equations, we have to establish some additional parameters. Let m be the mass of the pendulum, M the mass of the carriage, and J the moment of inertia of the pendulum with respect to its center of gravity.

The forces acting on the pendulum are the force of gravity, mg, acting on its center of gravity, a horizontal reaction force, H, and a vertical reaction force, V. Figure 2.6.ii depicts the forces acting on the pendulum and the cart. Taking moments about the center of gravity of the pendulum, we have

$$J\ddot{\theta}(t) = VL\sin\theta(t) - HL\cos\theta(t).$$

Summing all of the forces acting on the pendulum in the horizontal and vertical directions, we have

$$\begin{aligned} V - mg &= m\frac{d^2}{dt^2}(L\cos\theta(t)) \\ H &= m\frac{d^2}{dt^2}(z(t) + L\sin\theta(t)). \end{aligned}$$

Summing all of the forces acting on the cart, we have

$$u(t) - H = M\ddot{z}(t),$$

where $u(t)$ is the (control) input.

Since the task is to keep the pendulum upright, we will assume that θ and $\dot{\theta}$ will remain close to 0. On the basis of this assumption, we make the standard approximations, $\sin\theta \approx \theta$ and $\cos\theta \approx 1$, obtaining

$$\begin{aligned} mL\ddot{\theta}(t) + (m + M)\ddot{z}(t) &= u(t) \\ (J - mL^2)\ddot{\theta}(t) + mL\ddot{z}(t) - mgL\theta(t) &= 0 \end{aligned}$$

We introduce values for the remaining parameters.

$$M = 1\text{ kg},\ m = 0.15\text{ kg},\ L = 1\text{ m}$$

Using any mechanics or physics textbook, we get

$$\begin{aligned} g &= 9.81 \text{ m/sec}^2 \\ J &= \frac{4}{3} m L^2 = 0.2 \text{ kg-m}^2 \end{aligned}$$

Using these equations and parameter values, we obtain

$$\begin{aligned} 0.15\,\ddot{\theta}(t) + 1.5\,\ddot{x}(t) &= u(t) \\ 0.35\,\ddot{\theta}(t) + 0.15\,\ddot{x}(t) - 0.15 \times 9.81\,\theta(t) &= 0 \end{aligned}$$

to arrive at the following state and output equations for the dynamical model:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5809 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 4.4537 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 0.9211 \\ 0 \\ -0.3947 \end{bmatrix} \mathbf{u}(t) \\ &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{x}(t) \\ &= C\mathbf{x}(t) \end{aligned}$$

$\dot{x}(t) = \begin{bmatrix} \dot{x} \\ \ddot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$

where did the numb come from

position and angle

where we assume realistically that the only component of the output that is directly observable is the angle, θ, corresponding to the tilt of the missile in the case of the booster rocket.

In Chapter 4, we highlight results from linear systems theory that allow us to establish important properties (*e.g.*, stability and controllability) of dynamical systems, using simple tests on the matrices that define the state and output equations. The inverted pendulum is particularly interesting as it represents a dynamical system that is not stable, but is controllable.

Before leaving this chapter, we introduce some additional concepts and terms. We will develop similar concepts in the next chapter, in some cases using the same terms and in other cases introducing new terminology. Where the terminology differs, we will point out the conceptual similarities. An *event* is simply a pair consisting of a time point and a state (*e.g.*, $\langle t, x \rangle$ where $t \in T$ and $x \in X$). The event (or *phase*) space is the space of all possible events, $T \times X$.[6] A *state-space trajectory* is simply a mapping from

$\ddot{\theta} = f(\theta\ \dot{\theta}\ x)$

$\ddot{x} = g(\theta\ \dot{\theta}\ x)$

[6] We follow Kalman [9] in our use of the term phase space. You may also see the term used to refer to the space of possible positions and velocities. A state variable obtained from a system variable and its derivative is referred to as a *phase variable* [8].

the real interval to the state space, $h : [0, 1] \to X$, defined by a particular transition function, f, input, u, and initial conditions, $x(0) = x_0$. In the following chapter, we turn our attention to the use of logic in modeling physical systems.

2.4 Further Reading

For a general introduction to modeling from the perspective of control, see the texts by Dorf [6] or Bollinger [3]. For an emphasis on modern control, time-domain analysis, and, in particular, linear system theory, see Chen [5] or Gopal [8]. Our treatment of dynamical systems follows that of Kalman; Kalman's chapter in [9] provides a very general formulation of dynamical systems and an introduction to the necessary mathematical abstractions.

Bibliography

[1] Åström. Karl J. and Wittenmark. Björn, *Adaptive Control*. (McGraw-Hill. New York, 1989).

[2] Bellman. Richard. *Adaptive Control Processes*. (Princeton University Press, Princeton, New Jersey, 1961).

[3] Bollinger, John G. and Duffie. Neil A., *Computer Control of Machines and Processes*, (Addison-Wesley, Reading, Massachusetts, 1988).

[4] Borrie, John A., *Modern Control Systems: A Manual of Design Methods*, (Prentice-Hall, Englewood Cliffs, New Jersey, 1986).

[5] Chen. C.T., *Introduction to Linear System Theory*, (Holt, Rinehart, and Winston. New York, 1970).

[6] Dorf. Richard C., *Modern Control Systems*, (Addison-Wesley, Reading, Massachusetts, 1989).

[7] Gibson, John E., *Nonlinear Automatic Control*, (McGraw-Hill, New York. 1963).

[8] Gopal. M.. *Modern Control System Theory*, (Halsted Press. New York. 1985).

[9] Kalman, R. E., Falb. P. L., and Arbib, M. A., *Topics in Mathematical System Theory*, (McGraw-Hill, New York, 1969).

[10] Rabenstein, Albert L.. *Elementary Differential Equations with Linear Algebra*. (Academic Press. New York, 1975).

[11] Sastry, Shankar and Bodson. Marc, *Adaptive Control: Stability. Convergence, and Robustness*, (Prentice-Hall, Englewood Cliffs. New Jersey, 1989).

[12] Wolovich, William A., *Robotics: Basic Analysis and Design*, (Holt, Rinehart and Winston, 1987).

Chapter 3

Temporal Reasoning

Section 3.1 considers the use of temporal logic in reasoning about processes with an emphasis on the issues that arise in dealing with incomplete information. The temporal logic makes use of the differential calculus to reason about continuously changing parameters while at the same time providing precise semantics for reasoning about discontinuous change and incomplete information. In Section 3.2, we develop a computational language implementing many features of the temporal logic, and investigate some issues that arise in building practical systems for modeling processes.

3.1 Modeling Change in Temporal Logic

In this section, we consider methods for modeling physical systems based upon the first-order predicate calculus. We begin by identifying the sorts of entities that we need to reason about. Whereas the methods of the previous chapter focus on the *behavior of real-valued variables* over time, in this section the representations are designed primarily to facilitate reasoning about the *truth value of propositions* at various points in time. The propositions that we consider may correspond to statements about the value of real-valued variables, but we are not restricted to statements of that form.

There is a long history of calculi for reasoning about time in philosophy, computer science, and artificial intelligence. Rather than debate the advantages and disadvantages of the many existing techniques, we take the expedient of adopting a particular temporal logic that suits our basic needs for modeling physical systems. We then augment that logic to handle the

[0]©1990 Thomas Dean and Michael Wellman. All rights reserved.

s-ecific requirements of the applications considered in this monograph. In Section 3.3, we briefly consider some competing approaches to reasoning about time and provide references to papers dealing with complications not adequately addressed by our treatment.

To model physical processes, we need to reason about the truth of propositions over intervals of time. The propositions correspond to properties of the world that are subject to change over time. For instance, we might want to say something about whether or not a particular furnace is turned on at a particular time; to do so, we introduce a relation, on, and a constant, furnace17, denoting the furnace that we have in mind. Since the furnace is on at some times and off at others, the proposition, on(furnace17), must be interpreted differently with respect to different times. The temporal logic that we employ here is essentially a calculus for reasoning about the association between time intervals and propositions.

In the following, we choose to treat time points as primitive and reason about intervals in terms of points. Time points are denoted $\mathtt{t}$ or $\mathtt{t}i$, $i \in \mathbb{Z}$ (*e.g.*, $\mathtt{t1}$, $\mathtt{t2}$). Variables ranging over time points are denoted t or t_i, $i \in \mathbb{Z}$ (*e.g.*, t_1, t_2). Later when we incorporate our temporal notation into PROLOG, we will adopt standard PROLOG syntax and notate time variables as T or T*i*, $i \in \mathbb{Z}$ (*e.g.*, T1, T2). We introduce a binary relation, $\preceq$, on time points indicating temporal precedence. If $\mathtt{t1}$ and $\mathtt{t2}$ are time points, then $\langle \mathtt{t1},\mathtt{t2} \rangle$ is an interval. The formula $\langle\langle \mathtt{t1},\mathtt{t2} \rangle,\mathtt{p}\rangle$, where p is a propositional symbol, allows us to refer to the association between $\langle \mathtt{t1},\mathtt{t2} \rangle$ and p. Following common practice in artificial intelligence, we substitute holds(t1,t2,p) for $\langle\langle \mathtt{t1},\mathtt{t2} \rangle,\mathtt{p}\rangle$. The full specification of the syntax for the logic is described below.[1]

- *TC*: a set of time point symbols
- *C*: a set of constant symbols disjoint from *TC*
- *TV*: a set of temporal variables
- *V*: a set of variables disjoint from *TV*
- *TF*: a set of fixed-arity temporal function symbols
- *F*: a set of fixed-arity function symbols disjoint from *TF*

[1] The syntax for the first-order case and the semantics for the propositional case are borrowed directly from Shoham [58].

- R: a set of fixed-arity relation symbols
- $\preceq$: a binary relation symbol

The set of *temporal terms* (TT) is defined inductively as follows:

1. $(TC \cup TV) \subset TT$
2. If $trm_1 \in TT, \ldots, trm_n \in TT$, and $\mathtt{f} \in TF$ is an n-ary function symbol, then $\mathtt{f}(trm_1, \ldots, trm_n) \in TT$.

The set of *nontemporal terms* (NT) is defined similarly with TC replaced by C, TV replaced by V, and TF replaced by F.
The set of well-formed formulae (wffs) is defined inductively as follows:

1. If $trm_a \in TT$ and $trm_a \in TT$, then $trm_a = trm_b$ and $trm_a \preceq trm_b$ are wffs.
2. If $trm_a \in TT$ and $trm_b \in TT$, $trm_1 \in NT, \ldots, trm_n \in NT$, and $\mathtt{r} \in R$ is an n-ary relation symbol, then
$\mathtt{holds}(trm_a, trm_b, \mathtt{r}(trm_1, \ldots, trm_n))$ is a wff.
3. If φ_1 and φ_2 are wffs, then so are $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$.
4. If φ is a wff and $x \in (TV \cup V)$, then $\forall x\, \varphi$ is a wff.

We assume the standard definitions of $\vee$, $\supset$, $\equiv$, and $\exists$, and we make use of the following shorthand:[2]

$$\mathtt{holds}(t_1, t_2, \varphi_1 \wedge \varphi_2) \Rightarrow \mathtt{holds}(t_1, t_2, \varphi_1) \wedge \mathtt{holds}(t_1, t_2, \varphi_2)$$
$$\mathtt{holds}(t_1, t_2, \neg\varphi) \Rightarrow \neg\mathtt{holds}(t_1, t_2, \varphi)$$

and so on. Finally, since the structure of time is generally isomorphic to the integers or the reals, we assume that the addition and subtraction of temporal terms is well defined. For instance,

$$\forall t_1, t_2 ((t_2 - t_1) > 5_{\mathrm{min}}) \supset \mathtt{holds}(t_1, t_2, \varphi)$$

is meant to indicate that φ holds in any interval longer than five minutes.

By introducing appropriate relation and function symbols, we can develop notations for representing a variety of phenomena using the above syntax. For instance,

[2] Note that the left-hand sides are not well formed; hence, we use $\Rightarrow$ indicating a rewrite rule rather that $\equiv$ indicating logical equivalence.

$$\mathtt{holds}(\mathtt{t1},\mathtt{t2},\mathtt{temp}(\mathtt{room32})>72^\circ)$$

is meant to represent the fact that the temperature in a particular room is greater than 72° throughout the interval $\langle \mathtt{t1},\mathtt{t2} \rangle$. The following three formulae illustrate the use of quantification.

$$\begin{array}{l} \forall t_1,t_2 \\ \quad \mathtt{holds}(t_1,t_2,(\neg \mathtt{on}(\mathtt{furnace17}) \vee \mathtt{temp}(\mathtt{room32})>72^\circ)) \end{array}$$

$$\begin{array}{l} \forall t_1,t_2,r \\ \quad \mathtt{holds}(t_1,t_2,(\mathtt{on}(\mathtt{furnace17}) \wedge \mathtt{in\_room}(r,\mathtt{house32})) \supset \\ \quad \mathtt{holds}(t_1,t_2,\mathtt{temp}(r)>72^\circ) \end{array}$$

$$\begin{array}{l} \forall t_1,t_2,t_3,t_4 \; \exists t_5,t_6 \\ \quad ((t_1 \prec t_2 \prec t_3 \prec t_4) \wedge ((t_3 - t_2)>30_{\mathrm{min}}) \wedge \\ \quad\; \mathtt{holds}(t_1,t_4,\mathtt{temp}(\mathtt{outside})<20^\circ) \wedge \\ \quad\; \mathtt{holds}(t_2,t_3,\mathtt{temp}(\mathtt{room32})>70^\circ)) \supset \\ \quad ((t_2 \prec t_5 \prec t_6 \prec t_3) \wedge \mathtt{holds}(t_5,t_6,\mathtt{on}(\mathtt{furnace17}))) \end{array}$$

The first formula is meant to represent the fact that it is always the case that either the temperature in a particular room is greater than 72° or the furnace is not on. The second formula is meant to represent the fact that, whenever the furnace is on, all of the rooms in the house are above 72°. The third formula is meant to represent the fact that, if the temperature in a particular room is greater than 70° throughout an interval of greater than 30 minutes in length during which the outside temperature is less than 20°, then the furnace was on for some subinterval of duration 5 minutes or longer. There are also things that can not be represented in this logic. For instance, the logic is not powerful enough to represent the fact that the furnace was on for at least 5 minutes during a given 10 minute interval, where that 5 minutes could be spread out over an indeterminate number of subintervals.

We introduce some additional notations and conventions to simplify our notation. To simplify making statements about an assertion being true at a time point, we introduce the following abbreviation:

$$\forall t \; \mathtt{holds}(t,t,\varphi) \equiv \mathtt{holds}(t,\varphi)$$

It will frequently be useful to state that certain properties are timelessly true; for convenience, we define the "always" operator, □, as

$$\forall t_1,t_2 \; \mathtt{holds}(t_1,t_2,\varphi) \equiv \Box\varphi$$

' inally, we dispense with universal quantifiers that range over a textually isolated formula and assume that all free variables are universally quantified of scope the entire formula in which they are contained. For instance, in the following formula

holds(t_1,t_2,(¬on(furnace17) ∨ temp(room32)>72°))

we assume that the two temporal variables are universally quantified.

The two things that logicians are most concerned about in a logic is its proof theory and its semantics. Since we will not be concerned with proving theorems in the traditional sense, we will not bother with a proof theory for our logic. We are, however, concerned that our notations have precise meaning. Later, when we consider an algorithm for deriving statements from a set of other statements, we want to be assured that our conclusions are valid; for this, we require a semantic theory for our temporal logic.

Intuitively, the formula holds(t1,t2,on(furnace17)) should be true just in case the furnace is on at every time point between t1 and t2. In a modal logic, we can make that intuition concrete by thinking of time points as *possible worlds*. A possible world roughly corresponds to a model in traditional Tarskian semantics (*i.e.,* an assignment of true or false to each proposition). The different possible worlds are related to one another by the ordering relationship $\preceq$. In the first-order temporal logic presented here, we take a different approach to characterizing the meaning of formulae; we think of each proposition (*e.g.,* on(furnace17)) as denoting a set of time intervals. In this case, holds(t1,t2,on(furnace17)) should be true just in case ⟨t1,t2⟩ ∈ on(furnace17). To make this more precise, we provide the semantics for the propositional form of our temporal logic.[3]

The propositional case of our temporal logic is similar to the first-order case described above with the exception that there are no nontemporal variables, constants, or function symbols, and, instead of complex terms and relations, we have P a set of propositional symbols. In order to communicate the essential semantic properties of the logic, it should suffice to provide the semantics for the propositional case.

An *interpretation* is a triple $\langle TW, \leq, M\rangle$ consisting of a nonempty universe of time points, TW; a binary relation, $\leq$, on TW; and a two-part meaning function, $M = \langle M_1, M_2\rangle$, where $M_1 : TC \rightarrow TW$ and $M_2 : P \rightarrow 2^{TW \times TW}$.

[3] The propositional interval logic allows quantification over time points as in the first-order case, but is restricted so that φ in holds(t_1,t_2,φ) is a propositional formula.

A *variable assignment* is function $VA : TV \rightarrow TW$. If $u \in (TC \cup TV)$, we define $VAL(u)$ to be $M_1(u)$ if $u \in TC$, and $VA(u)$ if $u \in TV$. An interpretation $S = \langle TW, \leq, \langle M_1, M_2 \rangle \rangle$ is said to *satisfy* a wff φ under the variable assignment VA (written $S \models \varphi[VA]$) under the following conditions:

1. $S \models (u_1 = u_2)[VA]$ iff $VAL(u_1) = VAL(u_2)$
2. $S \models (u_1 \preceq u_2)[VA]$ iff $VAL(u_1) \leq VAL(u_2)$
3. $S \models \texttt{holds}(u_1, u_2, \varphi)[VA]$ iff $\langle VAL(u_1), VAL(u_2) \rangle \in M_2(\varphi)$
4. $S \models (\varphi_1 \wedge \varphi_2)[VA]$ iff $S \models \varphi_1[VA]$ and $S \models \varphi_2[VA]$
5. $S \models \neg\varphi[VA]$ iff $S \not\models \varphi[VA]$
6. $S \models (\forall v \varphi)[VA]$ iff $S \models \varphi[VA']$ for all VA' that agrees with VA everywhere except possibly on v.

An interpretation S is said to be a *model* for a wff φ (written $S \models \varphi$) if $S \models \varphi[VA]$ for all variable assignments VA. A wff is said to be *satisfiable* if it has a model, and a wff is said to be *valid* (written $\models \varphi$) if its negation is not satisfiable. We will have to augment the above semantics as we extend the logic to handle more complicated forms of inference, but the basic semantics relating temporal intervals and propositions will be retained.

In order to reason about processes, it is often natural to speak in terms of events that precipitate change in the world. For instance, the toggling of a switch corresponds to an event that has as a consequence changes in an electrical circuit. The occurrence of an event corresponds to a particular type of proposition holding over an interval. Shoham [58] provides a classification of proposition types that enables us to distinguish between those corresponding to the occurrence of events and those corresponding to other sorts of phenomena.

Most of the propositions that we have seen so far (*e.g.*, `on(furnace)`, `temp(room32)>70°`) are said to be *liquid* in Shoham's classification. A proposition type is liquid if, whenever it is is true over an interval, it is true over every subinterval (except possibly the endpoints), and, additionally, whenever it holds for all proper subintervals of some nonpoint interval (except possibly the endpoints), it holds over the nonpoint interval. Events are generally thought of as corresponding to propositions that are not liquid; they are said to be *gestalt* in Shoham's classification scheme. A proposition type is gestalt if, whenever it holds over an interval, it does not hold over any proper subinterval. To emphasize the role of events in reasoning about change, we use $\mathtt{occurs}(t_1, t_2, \varphi)$ instead of $\mathtt{holds}(t_1, t_2, \varphi)$ where φ is a gestalt proposition type corresponding to the occurrence of an event.

Suppose that the set of time points is isomorphic to the integers. For any given time point t, there exists a unique next time point $t+1$. We can specify a simple law of change as follows:

$$\begin{aligned}&\mathtt{R1:}\ (\mathtt{holds}(t, \neg\mathtt{on(furnace17)}) \wedge \\ &\qquad \mathtt{occurs}(t, \mathtt{toggle(switch42)})) \supset \mathtt{holds}(t+1, \mathtt{on(furnace17)})\end{aligned}$$

Of course, this rule is not quite right; the furnace does not always come on when you toggle its switch. *Use "axiom" instead of "rule." Indicate that what we really want is a weaker approximation of* R1, *but that we cannot provide such an approximation within the classical logic.* The fuse on the circuit feeding power to the furnace has to be intact, the furnace has to be mechanically and electrically sound, and any number of additional conditions must hold in order for the furnace to come on as a consequence of toggling its switch. Unfortunately, it generally will not be possible to enumerate all of the necessary conditions, and, even if you could enumerate them, the rule would be useless given that you could never know enough to establish whether or not all of the conditions are met in a given situation. The conditions specified in the antecedent of a rule such as R1 are meant to correspond to conditions that are readily known and usually sufficient to warrant the conclusion. The idea is that, if you frequently come to the right conclusion and only occasionally come to the wrong conclusion, then the small reduction in reliability will be offset by potentially enormous computational savings.

However, even if you are willing to accept the reduction in reliability that results from using R1, you may not be willing to accept another, more serious consequence of using rules of this form. The more serious consequence has to do with handling situations in which it is known that some necessary, but unaccounted for condition is not satisfied. For instance, you may know that the fuse on the circuit providing power to the furnace is open, rendering the switch useless. Unfortunately, the consequent of R1 still follows from the antecedent and you are left with a conclusion that you know to be false. What you would like to say is that the furnace will be on if you toggle its switch *unless* you have some information to the contrary. Formalizing this sort of inference is actually quite complex. The problem of reasoning about the conditions required for an event to have a given consequence is referred to as the *qualification problem* and is of considerable interest to researchers working in the area of default reasoning and nonmonotonic logic. We introduce some additional syntax that attempts to address the qualification problem as follows:

$$
\begin{aligned}
&\texttt{R2: (holds(}t\texttt{,}\neg\texttt{on(furnace17))} \land \\
&\qquad \texttt{occurs(}t\texttt{,toggle(switch42))} \land \\
&\qquad \neg\texttt{abnormal(R2,}t\texttt{))} \supset \texttt{holds(}t+1\texttt{,on(furnace17))}
\end{aligned}
$$

where abnormal(R2,t) is meant to indicate that R2 is inappropriate to apply with respect to t; in this case, R2 is said to be *disabled*. The status of the abnormal antecedent in R2 is different from that of the other two antecedents in the rule. The intent that the conclusion should follow as long as there is no evidence that the rule is abnormal. We can now add rules that will serve to disable R2 in appropriate circumstances. For instance,

$$
\begin{aligned}
&\texttt{Q1: (holds(}t\texttt{,open(fuse43))} \land \texttt{occurs(}t\texttt{,toggle(switch42)))} \supset \\
&\qquad \texttt{abnormal(R2,}t\texttt{)}
\end{aligned}
$$

indicates that the conclusion of R2 is not warranted whenever a certain fuse is open.

The intent behind R2 is that holds($t+1$,on(furnace17)) should follow from the axioms (*i.e.*, be a theorem) just in case holds(t,$\neg$on(furnace17)) and occurs(t,toggle(switch42)) follow, and $\neg$abnormal(R2,t) is consistent with the axioms. Unfortunately, if you use such a criterion to construct the set of theorems, you may get different answers depending upon the order in which you consider candidate formulae for membership in the set of theorems. In some cases, we can avoid ambiguity regarding the set of theorems by requiring that only a minimal number of abnormalities are allowed to occur. We can make our intended meaning precise by augmenting our semantic theory.

First, we introduce the idea of a partial ordering or *preference*, $\ll$, on models for a given set of axioms. Let Γ be the set of axioms describing how events precipitate change in the world. Γ would include rules such as R2, qualifications such as Q1, and additional axioms indicating initial conditions, observations, or proposed actions. We denote the set of all models of Γ (*i.e.*, $\{M : M \models \Gamma\}$) by $\mathrm{Mod}(\Gamma)$. Assuming that there are no infinite (descending) sequences of models $M_1, M_2, M_3 \ldots$ such that $M_2 \ll M_1, M_3 \ll M_2, \ldots$, the notion of the set of all minimal (with respect to $\ll$) models is well defined; we denote this set as $\mathrm{Min}(\ll, \mathrm{Mod}(\Gamma))$. We define a particular $\ll$ such that $M_1 \ll M_2$ just in case:

1. M_1 and M_2 agree on the interpretation of all function and relation symbols other than abnormal.

2. For all x and t, if $M_1 \models \mathtt{abnormal}(x,t)$, then $M_2 \models \mathtt{abnormal}(x,t)$.

3. There exists some x and t. for which $M_2 \models \mathtt{abnormal}(x,t)$, but $M_1 \not\models \mathtt{abnormal}(x,t)$.

We say that Γ *preferentially entails* φ with respect to $\ll$ (written $\Gamma \models_{\ll} \varphi$) just in case

$$\forall M \in \mathrm{Min}(\ll, \mathrm{Mod}(\Gamma)),\ M \models \varphi.$$

To illustrate, consider the following two observations:

O1: occurs(1,toggle(switch42))

O2: holds(1,¬on(furnace17))

indicating that the furnace was not on at time point 1, and that the switch was toggled at that time. Suppose that the set of axioms is

$$\Gamma = \{\mathtt{O1}, \mathtt{O2}, \mathtt{R2}, \mathtt{Q1}\}.$$

In this case, holds(2,on(furnace17)) is true in all models minimal with respect to $\ll$, and, hence, we have

$$\Gamma \models_{\ll} \mathtt{holds(2,on(furnace17))}.$$

Unfortunately, there are situations in which our augmented semantics runs counter to our expectations. For instance, suppose that we complicate our furnace scenario, and add a new rule indicating that, whenever a power surge occurs and we have no reason to believe that there are other complications, the fuse on the circuit providing power to the furnace overheats, leaving the circuit open.

R3: occurs(t,surge) ∧ ¬abnormal(R3,t) ⊃ holds($t+1$,open(fuse43))

In addition, suppose that we have observed a power surge at time 0.

O3: occurs(0,surge)

Given the set of axioms

$$\Gamma = \{\mathtt{O1}, \mathtt{O2}, \mathtt{O3}, \mathtt{R2}, \mathtt{R3}, \mathtt{Q1}\},$$

one might expect to conclude:

C1: holds(1,open(fuse43)) ∧ ¬holds(2,on(furnace(17))

However, while there are models minimal with respect to $\ll$ that satisfy C1, there are also minimal models satisfying:

C2: $\neg$holds(1,open(fuse43)) $\land$ holds(2,on(furnace(17))

It seems more plausible that evidence for an abnormality come from the past rather than from the future; hence, we should prefer models that allow us to conclude C1 over those that allow us to conclude C2. In general, we prefer models in which the fewest abnormalities occur, and those that occur do so as late as possible. The minimal models with respect to this preference are said to be *chronologically minimal*. We make this more precise by defining a new preference, $\ll_t$, such that $M_1 \ll_t M_2$ just in case there exists a time t such that:

1. M_1 and M_2 agree on the interpretation of all function and relation symbols other than abnormal.
2. For all x and $t' \prec t$, if $M_2 \models$ abnormal(x,t'), $M_1 \models$ abnormal(x,t').
3. For all x and $t' \preceq t$, if $M_1 \models$ abnormal(x,t'), $M_2 \models$ abnormal(x,t').
4. There exists some x, for which $M_2 \models$ abnormal(x,t), but $M_1 \not\models$ abnormal(x,t).

Given the set of axioms {O1,O2,O3,R2,R3,Q1}, C1 is true in all models minimal with respect to $\ll_t$.

The above discussion outlines some techniques for reasoning about what things change as a consequence of events occurring, but we haven't said anything about what things do not change. If you toggle the switch to the furnace, what happens to the color of the car in the garage? Presumably the color of the car remains the same as it was before you toggled the switch, but the axioms do not support this inference. We could provide an axiom like

R4: (holds(t,color(car45)) $\land$
    occurs(toggle(switch42))) $\supset$ holds($t+1$,color(car45))

but we would have to write a lot of axioms: one for each event/proposition pair,[4] and more if we are to account for combinations of events happening

[4] We would also have to add an "abnormal" condition as in R2 to handle that rare, but possible situation in which toggling the switch to your furnace somehow does change the color of your car.

at the same time. R4 is called a *frame xiom*, and the problem of reasoning about what things do not change as a consequence of an event occurring is called the *frame problem*.[5] In considering how to deal with the frame problem, we begin by considering the case in which time is modeled after the integers.

In the following, we attempt to augment our temporal logic so that propositions, once they become true, tend to persist in lieu of any information to the contrary. This augmentation is often referred to as the *default rule of persistence* [43], or the *common-sense law of inertia* [37]. The justification for adding this default rule is not based on any natural law. In fact, it does not appear to be appropriate for reasoning about propositions in general. We claim, however, that it is appropriate for reasoning about propositions describing many of the processes that we humans cope with on a day-to-day basis. This claim is based on an assessment of our perceptual and cognitive capabilities; we simply cannot cope with processes whose important properties are not discernible by our senses or that change so rapidly or seemingly randomly that we cannot keep track of them.

We begin by introducing a special case of abnormality. Since propositions tend to persist, times at which they change should be rare or abnormal. We refer to the abnormality in which a proposition φ changes its truth value at time t as a *clipping*, and notate it as $\mathtt{clips}(t,\varphi)$. *Note that there are problems with our treatment of clipping. In particular, the predicate* clips *ranges over other predicates. We took care to indicate that* $\mathtt{holds}(t_1,t_2,\varphi)$ *was just syntactic sugar for* $\langle\langle t_1,t_2\rangle,\varphi\rangle$, *but here we will probably just let it slide rather than get bogged down in complicated details.*

The following axiom schema allows us to infer clippings in appropriate circumstances:

$$\mathtt{AS1:}\ (\mathtt{holds}(t,\varphi) \wedge \mathtt{holds}(t+1,\neg\varphi)) \supset \mathtt{clips}(t,\varphi)$$

The common-sense law of inertia is captured in the following formula, which is logically equivalent to AS1:

$$\mathtt{AS2:}\ (\mathtt{holds}(t,\varphi) \wedge \neg\mathtt{clips}(t,\varphi)) \supset \mathtt{holds}(t+1,\varphi)$$

Since theorems of the form $\neg\mathtt{clips}(t,\varphi)$ generally do not follow from the axioms, for any t and φ, there will be models in which $\neg\mathtt{clips}(t,\varphi)$

[5] The name derives from the intuition that since very little changes from one frame to the next in a movie film, if you are told what does change, it should be simple to infer what does not [42].

is true and those in which it is false. We can use the same basi technique of minimizing temporally ordered abnormalities (*i.e.*, clippings in this case) that we used to deal with the qualification problem to ignore models with unwanted or unmotivated clippings. However, we have to be careful that clippings and other sorts of abnormalities do not interact in a counterintuitive manner. One way to control unwanted interactions between the two different sorts of abnormalities is to *prioritize* them using the following modification of $\ll_t$:

2'. For all x and $t' \prec t$, if $M_2 \models$ **abnormal**(x, t'), then $M_1 \models$ **abnormal**(x, t'), and, if $M_2 \models$ **clips**(x, t'), then $M_1 \models$ **clips**(x, t').

3'. For all x and $t' \preceq t$, if $M_1 \models$ **abnormal**(x, t'), then $M_2 \models$ **abnormal**(x, t'), and, if $M_1 \models$ **clips**(x, t'), then $M_2 \models$ **clips**(x, t').

4'. Either there exists some x, for which $M_2 \models$ **clips**(x, t), but $M_1 \not\models$ **clips**(x, t), or for all x, if $M_2 \models$ **clips**(x, t), then $M_1 \models$ **clips**(x, t), and there exists some x, for which $M_2 \models$ **abnormal**(x, t), but $M_1 \not\models$ **abnormal**(x, t).

Chronological minimization does not always perform according to our intuitions. To explain why not, we distinguish between two different sorts of temporal reasoning, referred to as *projection* and *explanation*.[6] Projection is the problem of reasoning forward in time from some initial state of affairs to determine the future course of events. Explanation is the problem of reasoning backward in time from some final state of affairs to determine the past course of events. Chronological minimization satisfies most of our intuitions regarding projection; unfortunately, it provides some rather counterintuitive results regarding explanation. For instance, suppose that the furnace is observed to be on at 9:00 in the evening and off at 8:00 the next morning. Chronological ignorance would have us conclude that the furnace was on all night and was turned off at the last possible moment before it was observed to be off at 8:00 AM. This inference strikes most as completely arbitrary, and is therefore an undesirable consequence of chronological minimization.

There has been a significant amount of work on designing a temporal logic that satisfies our intuitions regarding both projection and explanation,

[6] Here we assume the deterministic versions of these problems in which a specified initial [final] state of affairs uniquely determines the succeeding [preceding] course of events. Note that determinism in one direction does not necessarily imply the other.

and we will review this work briefly at the end of this section. Most of the deterministic problems that we consider in this book can be posed as projection problems of one sort or another. There is a real advantage to be had in casting a problem in terms of just projection or just explanation. In particular, the decision procedure used to automatically derive conclusions from a given axiomatic theory can exploit the (often linear) structure of time to expedite inference resulting in substantial computational savings. We return to deal with computational issues in Section 3.2.

Thus far, we have focused on modeling techniques that are suitable for reasoning about processes in which both time and change are discrete. While discrete modeling techniques provide suitable approximations for many continuous processes, we will find it convenient to extend our temporal logic to reason about continuous time and change. From now on, we assume that time is isomorphic to the reals. We have to reformulate the axiom schemata for dealing with the frame problem to handle continuous time.

$$
\begin{aligned}
\texttt{AS1'}:\ & ((t_1 \prec t \preceq t_2) \land \\
& \texttt{holds}(t_1,t,\varphi) \land \texttt{holds}(t,t_2,\neg\varphi)) \supset \texttt{clips}(t,\varphi)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{AS2'}:\ & ((t_1 \prec t \preceq t_2) \land \texttt{holds}(t_1,t,\varphi) \land \\
& \neg\exists t'((t \preceq t' \preceq t_2) \land \texttt{clips}(t',\varphi))) \supset \texttt{holds}(t,t_2,\varphi)
\end{aligned}
$$

In addition, our rules of change will look a bit different. For instance, we might change R2 to look like:

$$
\begin{aligned}
\texttt{R2'}:\ & ((t_1 \prec t) \land \texttt{holds}(t_1,t,\neg\texttt{on(furnace17)}) \land \\
& \texttt{occurs}(t,\texttt{toggle(switch42)}) \land \neg\texttt{abnormal}(\texttt{R2'},t)) \supset \\
& \exists t_2\ \ ((t+\epsilon) \prec t_2) \land \texttt{holds}(t+\epsilon,t_2,\texttt{on(furnace17)})
\end{aligned}
$$

where ϵ corresponds to a small delay between the time that the switch is toggled and the time that the furnace actually is on. This delay is meant to capture the intuition that causes precede effects. The delay is particularly appropriate here in that, were we to allow simultaneous cause and effect in this particular case, we would have an instant of time in which the furnace was both on and off.[7] *This has to be corrected. We still have the problem that the furnace is both on and off at some time.*

[7] This need not be true. We have not been careful to state whether or not our intervals $\langle t_1,t_2\rangle$ are closed, half open, or what. From our treatment of degenerate intervals (*e.g.*, $\langle t,t\rangle$), however, one might conclude that at least some intervals are closed. The additional notation and machinery necessary to resolve all of the issues concerning the status of time intervals is not deemed worthwhile for this discussion. We will continue to avoid such issues wherever possible, admitting that they would have to be resolved in a more complete treatment.

Say something more about the preference criterion for continuous time.

We will also find it useful to reason about quantities that change continuously as functions of time. Rather than invent new machinery within the interval temporal logic, we will try to import into the logic as much of the differential calculus as is needed for our anticipated control applications. Our treatment here roughly follows that of Sandewall [54].

First, we introduce a set, U, of real-valued parameters closed under the differential operator, ∂. If $u \in U$, then $\partial^n u \in U$, where $\partial^n u$ is the nth derivative of u with respect to time. We can trivially extend the syntax to represent statements about the values of parameters at various time points. For instance,

holds(t1,t2,$y = 3.1472$)

is meant to indicate that the parameter y has the value of 3.1472 throughout the interval ⟨t1,t2⟩. By restricting y to remain constant throughout the interval ⟨t1,t2⟩, we also restrict ∂y to remain 0 throughout the same interval.

To guarantee this intended meaning, we have to augment the semantics somewhat. In addition to a set of parameters U, we assume that each interpretation includes a function $Q : (\mathbf{R} \times U) \rightarrow \mathbf{R}$, where we employ the set of real numbers, $\mathbf{R}$, for the set of time points as well as for the set of all parameter values.

Since we will find it convenient on occasion to model abrupt changes in the value of parameters as they change over time, we introduce the notion of a *breakpoint*. We assume that a physical process is modeled using a set of differential equations that describe continuous changes in the parameters over intervals of time, and a set of axioms that determine what equations are appropriate over what intervals. Breakpoints are times at which the axioms signal a change in the differential equations used to model a given quantity or set of quantities. Generally, at a breakpoint, there is a discontinuity in some time-varying parameter.

We have to augment the semantics to account for the behavior of parameters with respect to breakpoints. Each interpretation must include a set of breakpoints $S \subset \mathbf{R}$, so that for all $u \in U$, $Q(t, u)$ is continuous over every interval not containing an element of S, and for all $t \not\in S$, $\frac{du}{dt} = Q(t, \partial u)$. Strange things can happen at breakpoints, but not so strange that we will allow a parameter to take on two different values. To avoid such anomalies, we will have to introduce some additional machinery.

At time t_0, we have a set of differential equations and a set of initial values[8] for all of the parameters; these equations and initial values are known to hold until some indeterminate time t_1, when a breakpoint occurs and the axioms determine a new set of differential equations and a new set of "initial" values. In order to establish breakpoints and the values for parameters immediately following breakpoints, we need to refer to the values of parameters "just before" and "just after" breakpoints. To do so, we define the left and right limits of a parameter x at time t as:

$$Q(t, x^l) \stackrel{\text{def}}{=} \lim_{\tau \to t^-} Q(\tau, x)$$

$$Q(t, x^r) \stackrel{\text{def}}{=} \lim_{\tau \to t^+} Q(\tau, x)$$

A *discontinuity* occurs at t with regard to a parameter x whenever the left and right limits are not identical:

$$Q(t, x^l) \neq Q(t, x^r)$$

As long as there are no discontinuities, the differential equations tell us exactly how the parameters vary with time. The axioms determine when breakpoints occur and what differential equations and initial conditions should be used to model processes between breakpoints. Discontinuities play a role in reasoning about real-valued quantities analogous to the role played by clippings in reasoning about the persistence of propositions. Just as the axioms do not rule out spurious models resulting from unexplained clippings, neither do they rule out models resulting from unexplained discontinuities.

Consider the following example. Suppose that we have two objects moving toward one another along a horizontal line. To keep the example simple, we assume that the surface is frictionless, the objects are represented as identical point masses, and there are no external forces acting on the objects. Let x_1 and x_2 represent the parameters corresponding to the position of the first and second objects, respectively, as measured from some reference on the horizontal line. At time 0, the first object is located at position 0, and the second object is located 10 meters to the right. A positive velocity indicates movement to the right. We make use of the standard notational conventions for position (x), velocity ($\partial x = \dot{x}$), and acceleration ($\partial^2 x = \ddot{x}$). Here are the axioms indicating the initial conditions:

[8] It is not necessary that the axioms establish the exact values for all parameters. The logic described here is well-suited to reasoning about inequalities and parameter ranges.

holds(0,$x_1 = 0$) holds(0,$x_2 = 10$)

holds(0,$\dot{x}_1 = 2$) holds(0,$\dot{x}_2 = -3$)

holds(0,$\ddot{x}_1 = 0$) holds(0,$\ddot{x}_2 = 0$)

where velocity is in units of meters per second. The next axiom determines the new velocities immediately following a collision breakpoint.

$$\Box((x_1 = x_2) \wedge ((\dot{x}_1 - \dot{x}_2) > 0)) \supset ((\dot{x}_1^l = \dot{x}_2^r) \wedge (\dot{x}_2^l = \dot{x}_1^r))$$

For the most part, the propositions corresponding to equations involving the parameters in U are constantly changing. In order for us to make useful predictions, however, certain equations have to persist over intervals of time. Suppose you are told that at time t_0, $x = 0$, $\dot{x} = 2$, and $\ddot{x} = 0$. If $x = 0$ persists, then there will be discontinuities in $\dot{x}$ and $\ddot{x}$. If $\ddot{x} = 0$ persists, then $\dot{x} = 2$ has to persist or be discontinuous in order to avoid a discontinuity in $\dot{x}$, and x is completely determined by $\dot{x} = 2$. However, if none of $x = 0$, $\dot{x} = 2$, or $\ddot{x} = 0$ persist, there need not be a discontinuity in any one of x, $\dot{x}$, or $\ddot{x}$, but neither is there any way of predicting the changes in x over time. In this example. we force an interpretation by stating that the accelerations for the two objects are always 0.

$$\Box((\ddot{x}_1 = 0) \wedge (\ddot{x}_2 = 0))$$

Using a preference analogous to $\ll_t$ that minimizes discontinuities, there is a single discontinuity in the acceleration of the objects two seconds after time 0, after which the objects, having exchanged velocities, head in opposite directions forever. We assume that the values of parameters are established in intervals not containing breakpoints by differential equations.

Note that, by our definition of clipping (*i.e.*, axiom schema AS1'), a discontinuity is a clipping only in the case that the discontinuity immediately follows a positive length interval in which the parameter is constant. We distinguish propositions corresponding to real-valued parameters taking on specific values (*e.g.*, $\ddot{x} = 2$) from propositions corresponding to truth-valued parameters (*e.g.*, on(furnace17)).

In the previous example, $\Box((\ddot{x}_1 = 0) \wedge (\ddot{x}_2 = 0))$ serves as the model for x_1 and x_2. In other cases, it may be convenient to infer a change in a model that persists over some indeterminate interval of time, just as we are able to infer changes in propositions that persist over intervals of time. To handle this sort of inference, we introduce a particular type of proposition pmod(x, m) where x is a real-valued parameter and m is a model for x. If

m is an nth-order differential equation, then it is assumed that the nth-order equation determines all higher-order derivatives, and all lower-order derivatives are known as part of the initial conditions. By stipulating $\Box(\ddot{x} = 0)$, we implicitly indicated $\texttt{holds}(0, \texttt{pmod}(x, \ddot{x} = 0))$ and that $x = 0$ and $\dot{x} = 2$ were the initial conditions at 0. Propositions of the form $\texttt{pmod}(x, m)$ persist according to chronological minimization. To illustrate how models might change over time, consider the following example.

Suppose that we want to reason about the temperature in a room heated by a furnace, and suppose that the furnace is controlled by a thermostat set to 70°. To make the example more interesting, suppose further that the thermostat has a 4° differential (*i.e.*, the furnace starts heating when the temperature drops to 68° and stops when the temperature climbs to 72°). To represent parameters "dropping to" or "climbing to" certain values, we define $\texttt{trans}([\downarrow \mid \uparrow], u, v)$ where $u \in U$ and $v \in \mathbf{R}$ as follows:

$$\begin{aligned}&\texttt{holds}(t, \texttt{trans}([\downarrow \mid \uparrow], u, v)) \equiv \\ &\qquad (Q(t,u) = v) \wedge (\exists t' \prec t, \forall t' \prec t'' \prec t, Q(t'', u)[> \mid <]Q(t,u))\end{aligned}$$

Propositions of the form $\texttt{trans}([\downarrow \mid \uparrow], u, v)$ are used to represent point events of the sort that trigger changes.

To model changes in the room's temperature when the furnace is off, we use Newton's law of cooling

$$\frac{dr}{dt} = -\kappa_1(r - a)$$

where r is the temperature of the room, a is the temperature outside the room, and κ_1 depends on the insulation surrounding the room. To model changes in the room's temperature when the furnace is running, we use

$$\frac{dr}{dt} = \kappa_2(f - r) - \kappa_1(r - a)$$

where f is the temperature of the furnace when it is running, and κ_2 depends on the heat flow characteristics of the furnace. The following axioms describe the temperature in the room over time.

$$\begin{aligned}&\Box(\texttt{trans}(\downarrow, r, 68°) \wedge \texttt{on}(\texttt{furnace17})) \supset \\ &\quad \texttt{pmod}(\texttt{r}, \partial r^r = -\kappa_1(r - a))\end{aligned}$$

$$\begin{aligned}&\Box(\texttt{trans}(\uparrow, r, 72°) \wedge \texttt{on}(\texttt{furnace17})) \supset \\ &\quad \texttt{pmod}(\texttt{r}, \partial r^r = \kappa_2(f - r) - \kappa_1(r - a))\end{aligned}$$

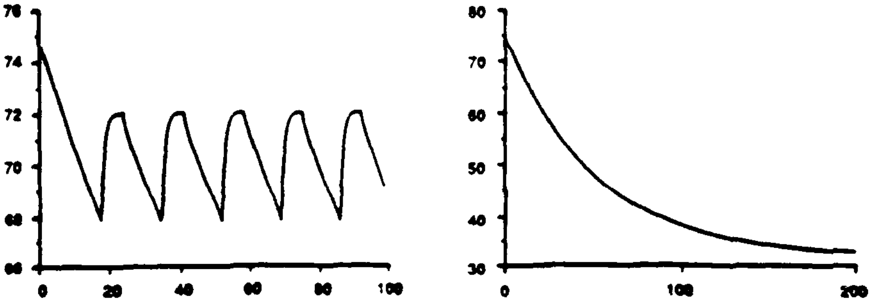


Figure 3.1: Different behaviors for a thermostatically controlled furnace

Suppose that we are interested in the temperature in the room over the interval from time 0 to time 10. We are told that the temperature outside is 32° throughout this interval, and that at time 0 the room is 75° with the furnace on but currently not heating. We represent these facts as follows:

holds(0,$r = 75^\circ$)

holds(0,10,$a = 32^\circ$)

holds(0,$\partial r = -\kappa_2(r - a)$)

$\exists t$ (0$\prec t$) $\wedge$ holds(0,t,on(furnace17))

We might expect the above axioms to support the following inferences. The temperature drops off exponentially[9] from 75° to 68° at which point the furnace starts heating and continues until the temperature reaches 72°, after which the furnace oscillates on and off forever with the temperature always between 68° and 72°. This expected behavior is shown on the left in Figure 3.1. Unfortunately, chronological minimisation of discontinuities does not support this inference. There are chronologically minimal models in which this is the case, but there are also chronologically minimal models in which on(furnace17) is clipped just at the time the temperature first drops to 68°, and instead of cycling forever between 68° and 72° the temperature in the room approaches 32° asymptotically as shown on the right in Figure 3.1.

We can eliminate the unintended models by not allowing simultaneous cause and effect. You can think of trans($[\downarrow \mid \uparrow], u, v$) events as a particular sort of causal trigger, and the propositions constraining parameters (*e.g.*,

[9] The behavior of the system can be described in terms of a piecewise continuous function in which the specific solutions for each piece are given, alternately, by $r(t) = 32^\circ + (r_0 - 32^\circ)e^{-\kappa t}$ and $r(t) = C + (r_0 - C)e^{-(\kappa_2+\kappa_1)t}$ where $C = \frac{\kappa_2 800^\circ + \kappa_1 32^\circ}{\kappa_1 + \kappa_2}$, r_0 is the initial temperature of the room for that particular piece and t is the time elapsed from the beginning of that piece.

$\partial r = \kappa(r - a)$) as a particular sort of effect. The general form of a *causal rule* is

$$\begin{array}{l} \texttt{holds}(t, [\textit{antecedent conditions}]) \land \\ \texttt{occurs}(t, [\textit{trigger event type}]) \land \\ \neg\texttt{abnormal}(t, [\textit{rule identifier}]) \supset \\ \exists t' \ ((t + \Delta) \prec t') \land \texttt{holds}(t + \Delta, t', [\textit{consequent effects}]) \end{array}$$

If $\Delta = 0$, then the antecedent conditions, the trigger event, and the consequent effects all compete with one another in the process of chronological minimization. Models in which the antecedent conditions are mysteriously clipped are equi-preferable to models in which the consequent effects occur as expected and result in clippings or discontinuities of their own.

Much of the work in temporal reasoning in artificial intelligence has focused on making precise the intuitions behind cause-and-effect reasoning. By requiring that causes precede effects, we not only avoid certain problems with unintended models, but we also subscribe to some of the basic intuitions about causal reasoning.

Our physical model for the thermostatically controlled furnace is not by any means complete. For instance, if we were to add the axiom

$$\exists t \ (8 < t) \land \texttt{holds}(8, t, \texttt{on(furnace17)})$$

we would arrive at the inappropriate conclusion that, if the furnace was heating at time 8, then it would continue to do so indefinitely. To avoid this unwanted inference, we might add rules saying that whenever anything results in the furnace "becoming" off, then the temperature in the room is governed by some default set of equations. To express this as an event triggered causal rule, we might define an analog of $\texttt{trans}([\downarrow \mid \uparrow], u, v)$ for truth-valued parameters. Suppose that $\texttt{becomes}(\varphi)$ corresponds to the event of φ becoming true. Adding the following axiom

$$\begin{array}{l} \texttt{holds}(t, \texttt{becomes}(\neg\texttt{on(furnace17)})) \supset \\ \exists t' (((t + \epsilon) \prec t') \land \texttt{holds}(t + \epsilon, t', \texttt{pmod}(\texttt{r}, \partial r^r = -\kappa_2(r - a)))) \end{array}$$

ensures that we will infer something reasonable in the event that the power to the furnace is cut off.

Note that we can always substitute a set of models that persist over different intervals of time for a single model that is true for all time but with additional parameters that make the model behave differently over different intervals of time. In the furnace example, we might state that

$$\Box(\partial r = \kappa_2(f - a) - \kappa_1(r - a))$$

and then have rules that govern the value of f over different intervals of time. Whether we vary the model or employ a single model and vary the parameters of the model, we have to provide some means for certain propositions corresponding to equations involving parameters to persist over time.

There remain many open issues in modeling physical systems using temporal logic that are not considered in this section. We will, however, return many times to consider both computational and representational issues in reasoning about time and change. In particular, the next section is concerned with automating temporal reasoning, Chapter 5 discusses how the temporal logic of this chapter can be used for planning, and Chapter 7 is concerned with temporal reasoning about stochastic processes.

Introduce the concepts of histories, time lines, chronicles and relate them to the notion of state-space trajectories introduced in the previous section.

3.2 Temporal Logic Programming

This section is concerned with the design of practical temporal reasoning systems. We describe a system that combines features from several existing systems to provide the support that we require for applications in planning and control. The resulting system is presented as an extension of the logic programming language PROLOG [9, 39] augmented with features, such as forward chaining, normally found in deductive retrieval systems [31].

In the last section, we presented a logic without regard to the complexity of determining whether or not a given formula was valid. Given that boolean satisfiability is *NP-complete* [21], we cannot expect to implement a decision procedure that is guaranteed to provide correct and timely answers to all possible queries. To ensure reasonable response time for our temporal reasoning system, we restrict the syntax for both queries and data. In addition, for some types of query, we provide only partial decision procedures (*i.e.*, procedures that occasionally report "don't know" in response to a query). This section represents a catalog of concessions to complexity. Completeness, expressiveness, and response time have to be carefully considered in the design of any program intended to serve as part of a control system. In Chapter 8, we consider tradeoffs in the design of decision procedures in some detail; in this section, we are primarily concerned with presenting the basic functions required for practical temporal reasoning, and pointing out potential sources of complexity.

For the most part, we adopt the syntax of PROLOG. Conditional rules (*i.e.*, PROLOG Horn clauses) are notated $A \leftarrow B$ where A is an atom (*i.e.*, a predicate of zero or more arguments) and B is a conjunction of zero or more atoms. We make use of the negation-as-failure operator, not, to implement various forms of nonmonotonic inference. (The query not(φ) succeeds just in case φ fails.) We assume the standard semantics for logic programs [3] augmented where needed with informal procedural semantics.

To speak about the structure of time itself, we refer to *points* (or *instants*) of time, and *intervals* (or *periods*) of time. We distinguish between a general type of event or proposition (*e.g.*, "I ate lunch in the cafeteria") and a specific instance of a general type (*e.g.*, "I ate lunch in the cafeteria this afternoon"). The latter are referred to as *time tokens* or simply *tokens*. A token associates a general type of event or proposition with a specific interval of time over which the event is said to occur or the proposition hold.

Our calculus for reasoning about time will be concerned with manipulating time tokens. Given some set of initial tokens corresponding to events and propositions, we will want to generate additional tokens corresponding to the consequences of the events. First, we have to be able to enter new tokens into the PROLOG database. We notate general types of events and propositions using PROLOG predicates and their negations. For instance, the proposition "the loading dock is unoccupied" might be represented as empty(loading_dock), and its negation as ¬empty(loading_dock). Similarly, the event type "truck #45 arrives at the loading dock" might appear as arrive(truck45,loading_dock). To enter a new token, we assert an expression of the form, token(*type*,*symbol*), where *type* corresponds to a general type of event or proposition, and *symbol* is a term that will be associated with an interval of time. Asserting

```
token(arrive(truck45,loading_dock),arrival14).
```

adds a new token of type arrive(truck45,loading_dock) and interval arrival14 to the database.

It is often convenient to refer to the points corresponding to the beginning and end of intervals. If arrival14 denotes an interval, then begin(arrival14) denotes its begin point and end(arrival14) denotes its end point. Initially, the interval of time associated with a token is completely unconstrained (*i.e.*, it could correspond to any interval). Intervals can be constrained using ordinal (*e.g.*, $\prec$ or $\preceq$) and metric constraints on their beginning and end points. If arrival14 and departure23 are both intervals, then asserting

```
end(arrival14)≺begin(departure23).
```

constrains the first interval to end before the second begins. For any interval, *int*, it is necessarily the case that

begin(*int*)$\preceq$ end(*int*).

Metric constraints allow us to bound the amount of time separating points. The notation distance(t_1,t_2) $\in$ [*low*,*high*] is used to specify that the distance in time separating t_1 and t_2 is bounded from above by *high* and bounded from below by *low*, where bounds are specified in the form, *hours:minutes*. For instance, if noon is a reference point corresponding to 12:00 PM today, asserting

distance(noon,begin(arrival14)) $\in$ [2:55,3:05].

constrains the interval associated with the arrival of truck45 to occur at 3:00 PM, give or take 5 minutes. If the upper and lower bounds are the same, we use $=$ instead of $\in$ and one number instead of a pair of numbers.

Given the *hours:minutes* notation for specifying metric constraints, we have committed to a set of time points isomorphic to $\mathbf{Z}$. We could have made it *hours:minutes:seconds*, but some concession ultimately has to be made to the finite precision of arithmetic on the target machine.

To indicate that a bound is unconstrained, we introduce the special symbol ∞, so that

- $\infty > n, \forall n \in \mathbf{Z}$
- $\infty + \infty = \infty + n = \infty, \forall n \in \mathbf{Z}$
- $\infty - \infty = 0$

Allowing both metric and ordinal constraints introduces some special problems in propagating (*i.e.*, combining) constraints to determine the *best bounds* on a pair of points (*i.e.*, the greatest lower and least upper bounds on the time separating the two time points). Propagation is simplified by adopting a single representation that captures both types of constraint. We do so by introducing yet another symbol ϵ with the following properties:

- $\epsilon > 0$
- $n * \epsilon < r, \forall n \in \mathbf{Z}, \forall r \in \mathbf{R}^+$
- $\epsilon + \epsilon = 2 * \epsilon > \epsilon$

Using the above, we define the following[10]

- $t_1 \prec t_2 \Rightarrow \texttt{distance}(t_1, t_2) \in [\epsilon, \infty]$.
- $t_1 \preceq t_2 \Rightarrow \texttt{distance}(t_1, t_2) \in [0, \infty]$.
- $t_1 = t_2 \Rightarrow \texttt{distance}(t_1, t_2) \in [0, 0]$.

We treat events and propositions somewhat differently in our calculus. We assume that the durations of events are specified precisely. For instance, we might state that the event corresponding to the arrival of truck45 took one minute.

```
distance(begin(arrival14),end(arrival14))=0:01.
```

For tokens corresponding to propositions, we would like to predict how long the propositions persist once they become true. For instance, suppose that were interested in reasoning about a robot forklift truck that moves appliances around in a warehouse, and suppose we make the following assertions to the database:

```
token(location(forklift,loading_area),location1).
token(location(forklift,staging_area),location2).
distance(noon,begin(location1))=1:15.
distance(noon,begin(location2))=2:30.
```

Assuming the forklift can only be in one of staging_area or loading_area, we conclude that the interval location1 should not persist past 2:30 PM. In general, we require that the interval corresponding to a token persist no further than the first subsequent interval corresponding to a token of a *contradictory type*. For any proposition type φ, φ and $\neg\varphi$ are said to be contradictory. Additional contradictory types have to be explicitly asserted. For instance, the assertion

```
contradicts(location(X,L1),location(X,L2)) ← L1≠L2.
```

indicates that any two tokens of type location(*arg1*,*arg2*) are contradictory if their first arguments are the same, and their second arguments are

[10] The constraints on time points are represented internally as pairs of complex numbers of the form $\langle\alpha,\beta\rangle$ for $\alpha + \beta\epsilon$, where $\alpha,\beta \in \mathbf{Z}$. For instance the bounds, $[\epsilon, 1]$ would be represented as $[\langle 0,1\rangle, \langle 1,0\rangle]$. The resulting calculus—first introduced by Leibnits [32] for studying the foundations of real analysis—provides a convenient basis for propagating and manipulating sets of equations including both ordinal and metric constraints.

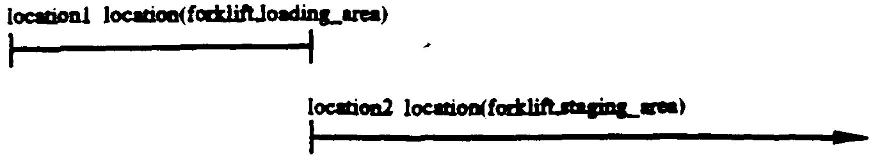

location1 location(forklift_loading_area)
location2 location(forklift_staging_area)

Figure 3.2: Tokens in the TEMPLOG database

different. The process of modifying the bounds on token intervals corresponding to propositions to ensure that tokens of contradictory types do not overlap is referred to as *persistence clipping*. One token is clipped by a second in accord with the following rule.

```
clips(K,begin(J)) ←
    token(P,K),
    token(Q,J),
    contradicts(P,Q),
    begin(K)≺begin(J).
```

The syntax for our temporal logic programming language severely restricts what can serve as a proposition type and what can be said about two different proposition types being contradictory. The consequences of these restrictions will become clearer as we explore the details query processing.

In the course of our discussions, we will be adding various capabilities to PROLOG to support applications in planning and control. We call this extended logic programming language TEMPLOG in recognition of the central role of time. For the time being, we assume that TEMPLOG automatically performs persistence clipping for all tokens stored in the database. Later we will have to relax this requirement to deal with the computational complexity of reasoning about partially ordered events.

It will help in this and subsequent chapters if we can display the contents of a TEMPLOG database graphically. To that end, we introduce the following graphical conventions. Time tokens are represented with a vertical bar indicating when the corresponding interval begins and either a second vertical bar providing some indication of when the interval ends or an arrow →, indicating that the end of the interval is far enough in the future that it can't be drawn in the diagram. The delimiters for tokens are connected by a horizontal bar (*e.g.*, ↦). Each token is labeled with a symbol corresponding to its associated interval and a formula denoting its type. The tokens are laid out on the page so as to indicate their relative offset from some global reference point. Figure 3.2 depicts the information stored in the TEMPLOG database as a consequence of the four assertions listed in the previous paragraph. In Figure 3.2, the token interval `location1` is constrained to end before the beginning of the token interval `location2` by the process of persistence clipping.

Given a database of time tokens, one is generally interested in answering queries concerning what propositions are true over what intervals of time. We begin by defining two primitive queries involving tokens and the bounds on the distance separating pairs of points. All of our other temporal queries can be defined in terms of these primitives.

- **token**(*type*, *int*) succeeds once for each token in the database unifying with *type* and *int*.

- **distance**(t_1,t_2) $\in [l,h]$ succeeds just in case $GLB \leq l \leq h \leq LUB$, where GLB and LUB correspond to the least upper and greatest lower bounds on the distance in time separating t_1 and t_2 given the closure of the set of constraints. If either t_1 or t_2 are not bound, the query will fail. If one or both of l and h are not bound, then, assuming that the query would succeed otherwise, it does so with the variables bound to their respective least restrictive bounds.

A *temporal query* of the form **holds**(t_1,t_2,φ), where φ is an atom, should succeed just in case there is a token in the database of type φ constrained to begin after or coincident with t_1 and not constrained to end before t_2. We can state this in terms of **token** and **distance** as follows.

```
holds(T1,T2,P) ←
    token(P,K),
    distance(begin(K),T1) ∈ [0,∞],
    not(distance(end(K),T2) ∈ [ε,∞].
```

and add an additional PROLOG rule to handle degenerate intervals

```
holds(T,P) ← holds(T,T,P).
```

Complex temporal queries involving conjunctions and disjunctions can be defined in terms of atomic queries using the standard PROLOG notational conventions (*i.e.*, `(P,Q)` and `(P;Q)` are, respectively, the conjunction and disjunction of `P` and `Q`). Conjunctive temporal queries are defined by

```
holds(T1,T2,(P,Q)) ← holds(T1,T2,P),holds(T1,T2,Q).
```

One way of defining disjunctive queries is

```
holds(T1,T2,(P;_)) ← holds(T1,T2,P).
holds(T1,T2,(_;Q)) ← holds(T1,T2,Q).
```

While this definition is simple to implement, it fails in some cases where we might expect it to succeed. For example, according to the definition above, if all we know is holds(t1,t2,p) and holds(t2,t3,q), holds(t1,t3,(p;q)) fails. As an alternative definition, we might have holds(t_1,t_2,(φ_1;φ_2)) just in case for all $t_1 \preceq t \preceq t_2$ either holds(t,φ_1) or holds(t,φ_2). The alternative definition does not, however, conform to the semantics of our logic of time intervals as given in the previous section; hence, we adopt the original definition from here on.

Using negation as failure, we can achieve some, but not all, of the functionality of true negation. For instance, we might define

```
holds(T1,T2,not(P)) ← not(holds(T1,T2,P)).
```

where not(holds(t_1,t_2,φ)) succeeds just in case holds(t_1,t_2,φ) fails.[11]

(Queries involving the negation-as-failure operator can be confusing to the uninitiated. As an example, the behavior of temporal queries in TEMPLOG involving unbound variables and the negation-as-failure operator is dependent upon the order of conjuncts just at it is for atemporal queries in PROLOG. For instance, assuming that holds(t1,t2,p(a)) and holds(t1,t2,q(b)), holds(t1,t2,(p(X),not(q(X)))) will succeed whereas holds(t1,t2,(not(q(X)),p(X))) will fail.)

While there is no direct mapping from negation in our logic to negation as failure in PROLOG, there are certain properties of the negation-as-failure operator that we might want to preserve in our temporal extensions of PROLOG queries. For instance, in PROLOG, not(not(φ)) succeeds if and only if φ succeeds. Note that holds(t_1,t_2,not(not(φ))) is (procedurally) equivalent to holds(t_1,t_2,φ) using the first definition but not using the second. We adopt the first definition in the following.

We assume that TEMPLOG processes both atomic and complex temporal queries efficiently. To illustrate TEMPLOG query processing, suppose that the following five queries are initiated in the database depicted in Figure 3.3.

```
holds(begin(service1),end(service1),
     location(truck45,loading_dock)).
```

[11] The two formulae holds(t_1,t_2,not(φ)) and holds(t_1,t_2,$\neg\varphi$) should not be confused. It is best to think of $\neg\varphi$ as a particular string defined to stand in some relationship to the string φ, where that relationship is defined by the operation of clipping. Alternatively, we might define holds(t_1,t_2,not(φ)) to succeed just in case there is some point t, such that $t_1 \preceq t \preceq t_2$ and holds(t,φ) fails.

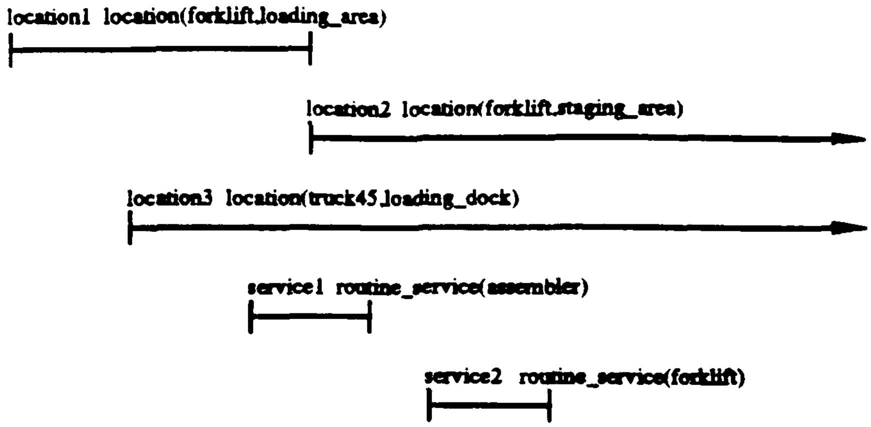

location1 location(forklift,loading_area)
location2 location(forklift,staging_area)
location3 location(truck45,loading_dock)
service1 routine_service(assembler)
service2 routine_service(forklift)

Figure 3.3: TEMPLOG database for illustrating query processing

```
holds(begin(service1),end(service1),
      (location(forklift,staging_area);
       location(truck45,loading_dock))).

holds(begin(service2),end(service2),
      (location(forklift,staging_area)),
       location(truck45,loading_dock))).

holds(begin(service2),end(service2),
      (location(Object,staging_area),
       location(Object,loading_dock)).

holds(begin(service2),end(service2),
      (location(Object1,staging_area),
       location(Object2,loading_dock)).
```

The first three queries succeed; the fourth fails, and the fifth succeeds with `Object1` bound to `forklift` and `Object2` bound to `truck45`.

There are also *abductive* versions of `holds` that are useful for building planning systems. The query `holds(`t_1`,`t_2`,`φ`)` fails if either of t_1 or t_2 are unbound. However, the abductive version of this query, $\Diamond$`holds(`t_1`,`t_2`,`φ`)`, succeeds under a superset of the conditions that `holds(`t_1`,`t_2`,`φ`)` does. In particular, if either t_1 or t_2 are not bound, then new (*i.e.*, totally unconstrained) points are created and bound to the variables. Once bound, the query succeeds if the set of constraints can be augmented so that the non-abductive query succeeds. The set of constraints necessary for the abductive query to succeed are referred to as *abductive constraints*. Abductive constraints are accumulated during backward chaining and withdrawn during backtracking similar to the way in which variable bindings are handled in PROLOG. Consider the database resulting from the following assertions.

```
token(p,j).                   distance(begin(j),end(j))=5.
token(q,k).                   distance(begin(j),begin(k))=3.
distance(t1,t2)=3.            distance(begin(k),t1) ∈ [-5,5].
```

Of the following six queries, those on the left fail while those on the right succeed.

```
holds(t1,t2,p).               ◇holds(t1,t2,p).
holds(t1,t2,q).               ◇holds(t1,t2,q).
◇holds(t1,t2,(p,q)).          ◇holds(t1,T2,(p,q)).
```

We will say more about abductive query processing in Chapter 5.

Persistence clipping is one type of routine inference important in reasoning about time and change. There is a second type of routine inference, called *projection*, that we would like TEMPLOG to perform for us. Projection is concerned with inferring the consequences of events based on a model specified in terms of the cause-and-effect relationships that exist between various event types. To notate such relationships, we use the following form

project(*antecedent_conditions*, *trigger_event*, *delay*, *consequent_effects*)

to indicate that, if an event of type *trigger event* occurs, and the *antecedent conditions* hold at the outset of the interval associated with *trigger event*, then the *consequent effects* are true after an interval of time determined by *delay*. The trigger event is specified as a type, the antecedent conditions and consequent effects are specified as types or conjunctions of types, and the delay is specified as a pair consisting of a lower and an upper bound on the time between the end of the trigger event and the manifestation of the effects. If the upper and lower bounds are the same, a single bound can be substituted for the pair. We assume a convenient notational filter so that the delay argument can be left out of assertions and queries; in the former case, a default delay of $[\epsilon,\epsilon]$ is provided. The rule R2 from the previous section can be encoded as follows.

```
project(¬on(furnace17),toggle(switch42),on(furnace17))
```

To specify that, whenever the forklift moves from one location to another, it will appear in the new location after a delay determined by the (spatial) distance to be traveled and the minimum and maximum rate of travel allowed by the forklift, we would assert the following

```
project(location(forklift,Loc1),
        move(Loc1,Loc2),
        [(distance(Loc1,Loc2) ÷ max_speed),
         (distance(Loc1,Loc2) ÷ min_speed)],
        location(forklift,Loc2)).
```

As another example, suppose that the robot forklift is also responsible for installing options in appliances (*e.g.*, installing an ice maker in a stock refrigerator). The following projection stipulates that whenever the robot turns on a particular assembly unit when an appliance and an appropriate option are on the input conveyor, then 30 minutes later, give or take five minutes, the appliance will appear in the output conveyor with the option properly installed.

```
project((status(assembler,off),
         location(Appliance,in_conveyor),
         instance_of(Appliance,home_appliance),
         location(Option,in_conveyor)
         instance_of(Option,option_for(Appliance))),
        push_button(on),
        [00:25,00:35],
        (installed(Appliance,Option),
         location(Appliance,out_conveyor),
         part_of(Option,Appliance))).
```

In order to determine whether or not an event has an effect at a particular time, we define the following

```
causes(E,R,T) ← project(P,E,R),occurs(E,T),holds(T,P).
```

The projection rules presented above allow for a very restricted form of causal reasoning. In particular, they do not provide for any means of dealing with the qualification problem described in the previous section. By modifying our causes rule slightly, we can reason about qualifications in a manner similar to that described in the previous section.

```
causes(E,R,T) ←
    project(P,E,R),
    occurs(E,T),holds(T,P),
    not(abnormal(E,R,T)).
```

The rule Q1 from the previous section can be encoded as follows.

```
abnormal(toggle(switch42),on(furnace17),T) ←
    holds(T,open(fuse43))
```

We include the type of the trigger event and the type of the consequent effect because the qualification is likely to depend on them. Note that neither is sufficient alone, since the event of toggling the switch may have other effects (*e.g.*, the switch may make a noise whether or not it makes or breaks a connection), and other events may have the effect of turning the furnace on (*e.g.*, attaching it directly to a backup diesel generator that bypasses the fused circuit). For more complicated applications, it may be useful to allow disabling rules that serve to disable other disabling rules. We do not do so here, but it would be straightforward to extend the above to handle a hierarchy of disabling rules (*i.e.*, a set of disabling rules arranged hierarchically with a projection rule at the root so that each disabling rule in the tree is allowed to disable its immediate ancestor in the tree).

Qualifications in projection rules allow us to introduce a very restricted form of quantification. As an example, consider the following rule.

```
project((clear(X),clear(Y),on(X,_)),puton(X,Y),on(X,Y)).
```

For an event of type puton(block1,block2) to have the consequent effect on(block1,block2), there have to be tokens in the database of type clear(block1) and clear(block2). Alternatively, we can use the following projection rule

```
project(on(X,_),puton(X,Y),on(X,Y)).
```

coupled with the following qualification

```
abnormal(puton(X,Y),on(X,Y),T) ← holds(T,(on(_,X);on(_,Y))).
```

to ensure that puton(block1,block2) has the effect on(block1,block2) just in case there are no tokens in the database with appropriately constrained intervals corresponding to something being on either block1 or block2.

Projection is the process of generating new tokens from some set of initial tokens—roughly corresponding to the boundary conditions in a physics problem—using a set of projection rules. The basic algorithm for handling both projection and persistence clipping is rather simple to implement. To simplify its description, we assume that all trigger events are point events. Whenever tokens or constraints are added to or deleted from the database, the system carries out the following steps.

1. Delete all tokens and constraints added the last time the algorithm was run.

2. Place all tokens in the database whose types correspond to events on the *open list*.

3. Let *token* be the earliest occurring token in the open list.

4. Find all rules whose trigger event type unifies with the type of *token*.

5. For each rule found in Step 4 whose antecedent conditions are satisfied, add to the database tokens corresponding to the types specified in the consequent effects, and constrain them according to the specified delay.

6. For each new token added in Step 5 whose type corresponds to an event, place it on the open list.

7. For each new token added in Step 5 whose type does not correspond to a fluent, find all tokens of a contradictory type that begin before the newly added token and constrain them to end before the beginning of the new token.

8. Remove *token* from the open list.

9. If there are no tokens remaining on the open list, then quit, else go to Step 3.

We will assume that TEMPLOG uses an algorithm similar to the above to ensure that the database contains all and only those tokens warranted by the set of initial tokens, and the projection rules stored in the database. Updates can be performed in time polynomial in the size of the initial conditions and the set of projection rules. Query processing is performed by searching through the set of tokens generated by the projection algorithm, using the types of the tokens and the constraints on token intervals to guide the search. The above projection algorithm supports basic reasoning about the truth or falsity of propositional formulae; in the following, we consider extensions to handle real-valued parameters.

Let U be a set of real-valued parameters, and P be a set of boolean-valued propositional variables.[12] In addition, we introduce two mappings $Q : \mathbf{R} \times U \rightarrow 2^{\mathbf{R}}$ and $V : \mathbf{R} \times P \rightarrow 2^{\{true, false\}}$. The task of projection is to determine Q and V for some closed interval of $\mathbf{R}$. We begin by considering the completely determined case in which both Q and V map to singleton sets (*i.e.*, $Q : \mathbf{R} \times U \rightarrow \mathbf{R}$ and $V : \mathbf{R} \times P \rightarrow \{true, false\}$).

[12] Due to the presence of variables and complex terms, templog rules are *schemata* for propositional axioms. The underlying logic remains purely propositional.

At the initial time point, we assume that the values of all parameters and propositional variables are known. In addition, we are given a set of events specified to occur at various times over the time interval of interest. We assume a set of projection rules as before. In addition, we assume a set of modeling rules for parameters in U. A modeling rule is just a special sort of projection rule; the basic form is the same as that introduced earlier in this section, the only difference being that the delay is always assumed to be ϵ, and the consequent effects consist of parameter assignments in the form of ordinary differential equations[13] with constant coefficients (*e.g.*, $\partial u = 2$ or $\partial^2 u = 3\partial u + 5u + 4$).

The projection rule from the last section for reasoning about the temperature of the room in the case that the furnace is off is encoded as follows.

$$\texttt{project(on(furnace17),trans(}\uparrow, r, 68^\circ\texttt{),pmod(}r, \partial r^r = -\kappa_1(r-a)\texttt{)))}.$$

To make sure that persistence clipping is handled correctly, we state that a given parameter can have only one model at a time.

$$\texttt{contradicts(pmod(X,M1),pmod(X,M2))} \leftarrow \texttt{M1} \neq \texttt{M2}.$$

Now we can state the basic algorithm for performing projection given some set of initial conditions and a projection interval $[t_s, t_f]$. To simplify the description of the algorithm, we assume that all events are point events (*i.e.*, if e is a type corresponding to the occurrence of an event, $\texttt{token}(e,k) \supset (\texttt{begin}(k) = \texttt{end}(k))$), and all events described in the initial conditions begin after t_s. Let $\mathcal{A}$ be the set of all currently active process models (*i.e.*, all m such that $\texttt{holds}(t_c, \texttt{pmod}(x,m))$ for some x). Let $\mathcal{E}$ be the set of pending events (*i.e.*, the set of all events, $\texttt{token}(e,k)$, generated so far such that $t_c \prec \texttt{begin}(k)$). Let $\mathcal{C}$ be the set of current conditions (*i.e.*, all $u^r = v$ such that there exists $m \in \mathcal{A}$ such that $\texttt{holds}(t_c, \texttt{pmod}(x,m))$, $u = \partial^n x$ for some n, and $\texttt{holds}(t_c, u^r = v)$.

In the cases that we are interested in, we can recast a set of ordinary differential equations and their initial conditions as a system of first-order differential equations. We can then solve these equations using numerical methods based on the Taylor expansion (*e.g.*, the Runge-Kutta methods [50]) and various forms of linear and nonlinear extrapolation (*e.g.*, the Adams-Bashforth and Adams-Moulton methods [56, 46]). The particular

[13] To expedite the necessary computations, we assume that all equations are 5th order or less, and that they can be rewritten so that highest-order term is algebraically isolated on the left-hand side of the equation.

numerical method chosen is not important for our discussion. In the following, we simply assume the ability to generate solutions to ordinary differential equations efficiently, and refer to the procedure for generating such solutions as the *extrapolation procedure*. Given a set of initial conditions and a projection interval $[t_s, t_f]$, projection is carried out by the following algorithm.

1. Set t_c to be t_s.

2. Set $\mathcal{E}$ to be the set of events specified in the initial conditions.

3. Using $\mathcal{A}$, $\mathcal{C}$, and the extrapolation procedure, find t_n corresponding to the earliest point in time following t_c such that the trigger for some projection rule is satisfied or t_f whichever comes first. If $t_n \neq t_f$, then t_n could be the time of occurrence of the earliest event in $\mathcal{E}$, or it could be earlier, corresponding to the solution of a set of equations (*e.g.*, $((x_1 = x_2) \wedge ((\dot{x}_1 - \dot{x}_2) > 0))$).

4. If $t_n = t_f$, then quit, else set t_c to be t_n.

5. Find all projection rules with the trigger found in Step 3.

6. For each rule found in Step 5 whose antecedent conditions are satisfied, add to the database tokens corresponding to the types of the consequent effects except in the case of consequent effects corresponding to parameter assignments (*e.g.*, $x_1^r = x_2^r$). Constrain the new tokens according the delay specified in the corresponding rule.

7. For each token added in Step 6 whose type corresponds to an event, add it to $\mathcal{E}$.

8. For each token added in Step 6 whose type does not correspond to an event, find all tokens of a contradictory type that begin before the newly added token and constrain them to end before the beginning of the new token.

9. If the trigger found in Step 3 corresponds to the type of an event token in $\mathcal{E}$ whose time of occurrence is t_c, remove it from $\mathcal{E}$.

10. Use the consequent effects corresponding to parameter assignments found in Step 6 and the results of extrapolation to determine $\mathcal{C}$. The parameter assignments corresponding to the consequent effects of projection rules take precedence over the extrapolation results.

11. Go to Step 3.

There are lots of other rules that we would have to specify in order to model the operation of the assembler in enough detail to support useful prediction. We would have to state that pushing the on button when the assembler is off causes it to become on,

```
project(status(assembler,off),
        push_button(on),
        status(assembler,on)).
```

and that a machine can not be on and off at the same time,

```
contradicts(status(X,S1),status(X,S2)) ← S1≠S2.
```

In fact, there are potentially an infinite number of rules that would be required to correctly model the behavior of the assembler under every set of circumstances. Note that the assembler requires power, and the appliance and the options to be installed must be in some reasonable state of repair, and there can't be anything blocking the output conveyor; all of these conditions and more would have to be made explicit in the rules if we required a model guaranteed to produce correct predictions in every conceivable situation. This proliferation of antecedent conditions was addressed in the context of the qualification problem discussed in Section 3.1. There is also a problem with consequent effects; If the robot places a part in a box, then the part is in the box. If the robot then places the box in a truck, then the part is still in the box, but it is also in the truck. If the robot then drives the truck to a new location, then, by virtue of being in the box which is in the truck, the part is in the new location also. Keeping track of all of the consequences of an action has been termed the *ramification* problem [18], and constitutes a significant problem in building practical temporal reasoning systems.

The TEMPLOG rules that comprise a physical model are intended as an approximation. Greater accuracy can often be obtained by adding more rules, but there is a price to be paid in terms of computational overhead, and the increased accuracy may not result in a significant increase in performance. The idea behind causal modeling is that an appropriate model will efficiently generate those common-sense predictions that are likely to have the greatest impact on the performance of the robot. It is up to the programmer to determine what rules are necessary to generate these common-sense predictions.

This is where the material on reasoning about partial orders and uncertainty should go (separate section?). What if the initial conditions are not *exact, but, rather, are specified in terms of intervals or distributions. Talk about the use (and abuse) of Monte-Carlo methods for reasoning about underspecified initial conditions. Introduce the notion of possible time lines, and connect this with model theory developed in Section 3.1. Finally, motivate the uncertainty issues developed in Chapter 7.*

3.3 Further Reading

Perhaps the best known approach to reasoning about change in artificial intelligence is the *situation calculus*. [40, 42, 34]. McCarthy is generally given credit for the basic idea, but many researchers have contributed to the development of what today is referred to as *the* situation calculus. A *situation* corresponds to the state of the world at a particular instant in time. Change results as a consequence of *actions* occurring in situations, where an action can be thought of as a function from situations to situations that maps the situation in which the action occurs into the *next* situation. While some attempts have been made to incorporate reasoning about continuous processes within the situation calculus [30], many researchers have considered other approaches for reasoning about real-world processes.

In the late 1970's, Hayes issued a challenge to the research community to formalize a large corpus of knowledge about physical processes [28]. Hayes got things started by proposing an axiomatic theory of how liquids behave [29]. Hayes's theory describes change over time using four-dimensional pieces of space-time called *histories*. Other researchers, interested in reasoning about physical phenomena whose spatial properties are less central, adopt a variety of temporal logics in which change is modeled in terms of some form of causal relation [2, 43]. The frame problem appeared in all of these logics in one form or another and some researchers believed that the frame problem could be solved by employing some form of nonmonotonic reasoning [41, 52, 44].

This belief that nonmonotonic reasoning would solve the frame problem was dealt a blow by the work of Hanks and McDermott, which showed that a straightforward application of existing nonmonotonic logics was not sufficient to solve the problem [24, 25]. The research community immediately countered with several proposals for solving the particular temporal reasoning problem posed by Hanks and McDermott [36, 33, 57], all based on some variation on the idea of chronological minimization. Subsequent work has focused on formalizing causation to solve the frame problem [37, 27], and coping with problems that involve reasoning both forward (projection) and backward (explanation) in time [45, 38, 4, 55]. *Say something about the possible worlds approach to reasoning about actions [22, 63].*

The idea of preferring certain models over others in order to define a notion of semantic entailment for nonmonotonic logics is due to Bossu and Siegel [6] and (independently) Shoham [58]. Shoham's formulation is the more general of the two. The idea of selecting models that are minimal with respect to some property and some ordering relation is developed in Lifschitz [36], Kautz [33], and Shoham [57]. The term "chronological minimization" is due to Shoham [57]. See also Doyle and Wellman [16] on some fundamental limitations of nonmonotonic logics based on preference orders.

Much of the work in the philosophical literature has focused on the use of modal logics to model time [49, 53, 59]. This has also been the case for theoretical computer science in designing logics to reason about computational processes [26, 48, 19, 47]. In the case of computer science, one important reason for the emphasis on modal logic is that such logics are somewhat easier to analyze in terms of the complexity of their respective decision problems. As far as expressive power is concerned, given that it is possible to translate any modal logic with standard Kripke semantics into classical logic, it would seem that the interval logic presented here is at least as expressive as any modal logic of time [58, 59].

The syntax and semantics for the propositional case of the temporal logic that we adopt were introduced to the artificial intelligence community by McDermott [43]. Shoham [58] provided the semantics for the first-order case, and it is a syntactic variant of his formulation that we use here.

There has been a significant amount of work in artificial intelligence on modeling physical processes without employing the sort of quantitative analysis prevalent in engineering. This work, involving *qualitative* reasoning about physical systems generally makes use of discrete value spaces and a special type of differential equation to draw conclusions about the behavior of continuous processes [5]. Given that the applications that we consider in this monograph typically require some sort of quantitative analysis, it seems reasonable to incorporate into our logic those parts of the differential calculus that seem made for the job [51]. The semantic treatment presented here is based on the work of Sandewall [54], but the basic approach to reasoning about processes was influenced significantly by the work of Forbus [20] and de Kleer [11].

The practical problems in building useful temporal reasoning systems are manifold, and have given rise to a rich technical literature. Much of the early work makes use of the situation calculus. Green describes a method for applying automated theorem proving to reasoning about time in the situation calculus [23]. Later work sought to avoid the need for frame axioms by introducing some form of nonmonotonic inference into the operation of the temporal reasoning algorithm. Fikes *et al.*} *implicitly make use of the common-sense law of inertia in their implementation of* STRIPS *[17]. The temporal reasoning system described in this section is based on the work of Dean [15, 12], but was influenced significantly by other event-based approaches to reasoning about time and causality* (e.g., *[1, 35, 60, 61])*.

Davis discusses the computational issues involved in propagating metric constraints for reasoning about time [10], and Dean considers the issues involved in organizing large amounts of temporal information so as to expedite the sort of causal reasoning described in this section [13]. Wilkins provides a wealth of practical advice for systems designers building the temporal reasoning component of a planning system; in particular, his discussion regarding the limited use of quantifiers in causal rules is worth reading [62]. It should be mentioned that the simple projection algorithm described above is not guaranteed to work properly if the tokens corresponding to the initial conditions are partially ordered. The general problem of predicting the consequences of a set of partially ordered events is potentially intractable [8]. To deal with this potential source of complexity, partial decision procedures have been developed to avoid expending too much effort in performing projection [14].

Bibliography

[1] *James Allen. Maintaining knowledge about temporal intervals.* Communications of the ACM, *26:832–843, 1983.*

[2] *James Allen. Towards a general theory of action and time.* Artificial Intelligence, *23:123–154, 1984.*

[3] *Krzysztof Apt and M.H. van Emden. Contributions to the theory of logic programming.* Journal of the ACM, *29:841–862, 1982.*

[4] *Andrew B. Baker and Matthew L. Ginsberg. Temporal projection and explanation. In* Proceedings IJCAI 11. *IJCAI, 1989.*

[5] *Daniel G. Bobrow, editor.* Qualitative Reasoning and Physical Systems. *MIT Press, Cambridge, Massachusetts, 1985.*

[6] *G. Bossu and P Siegel. Saturation nonmonotonic reasoning, and the closed-world assumption.* Artificial Intelligence, *25:173–183, 1985.*

[7] *Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors.* Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning. *Morgan-Kaufmann, Los Altos, California, 1989.*

[8] *David Chapman. Planning for conjunctive goals.* Artificial Intelligence, *32:333–377, 1987.*

[9] *W.F. Clocksin and C.S. Mellish.* Programming in Prolog. *Springer-Verlag, New York, 1984.*

[10] *Ernest Davis. Constraint propagation with interval labels.* Artificial Intelligence, *32:281–331, 1987.*

[11] Johan de Kleer. Multiple representations of knowledge in a mechanics problem solver. In Proceedings IJCAI 5, pages 299–304. IJCAI, 1977.

[12] Thomas Dean. An approach to reasoning about the effects of actions for automated planning systems. Annals of Operations Research, 12:147–167, 1988.

[13] Thomas Dean. Using temporal hierarchies to efficiently maintain large temporal databases. Journal of the ACM, 36(4):687–718, 1989.

[14] Thomas Dean and Mark Boddy. Reasoning about partially ordered events. Artificial Intelligence, 36(3):375–399, 1988.

[15] Thomas Dean and Drew V. McDermott. Temporal data base management. Artificial Intelligence, 32(1):1–55, 1987.

[16] Jon Doyle and Michael P. Wellman. Impediments to universal preference-based default theories. pages 94–102, San Mateo, CA, 1989. Morgan-Kaufmann.

[17] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2:189–208, 1971.

[18] Joseph Jeffrey Finger. Exploiting constraints in design synthesis. Ph.D. Thesis, Stanford University, 1987.

[19] Michael Fischer and Richard Ladner. Propositional modal logic of programs. In Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, pages 286–294, 1977.

[20] Kenneth D. Forbus. Qualitative process theory. In Bobrow [5], pages 85–168.

[21] Michael R. Garey and David S. Johnson. Computing and Intractibility: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York, 1979.

[22] Matthew L. Ginsberg and David E. Smith. Reasoning about action i: a possible worlds approach. Artificial Intelligence, 35:165–195, 1988.

[23] Cordell C. Green. Application of theorem proving to problem solving. In Proceedings IJCAI 1, pages 219–239. IJCAI, 1969.

[24] Steve Hanks and Drew V. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In Proceedings AAAI-86, pages 328–333. AAAI, 1986.

[25] Steve Hanks and Drew V. McDermott. Nonmonotonic logic and temporal projection. Artificial Intelligence, 33:379–412, 1987.

[26] David Harel. First-Order Dynamic Logic. Springer-Verlag, New York, 1979.

[27] Brian Haugh. Simple causal minimizations for temporal persistence and projection. In Proceedings AAAI-87, pages 218–223. AAAI, 1987.

[28] Patrick Hayes. The naive physics manifesto. In Donald Michie, editor, Expert Systems in the Microelectronic Age, pages 242–270. Edinburgh University Press, 1979.

[29] Patrick J. Hayes. Naive physics i: Ontology for liquids. In Jerry E. Hobbs and Robert C. Moore, editors, Formal Theories of the Common Sense World, pages 71–107. Ablex, Norwood, New Jersey, 1985.

[30] Gary Hendrix. Modeling simultaneous actions and continuous processes. Artificial Intelligence, 4:145–180, 1973.

[31] Carl Hewitt. Planner: A language for proving theorems in robots. In Proceedings IJCAI 1, pages 295–301. IJCAI, 1969.

[32] Albert E. Hurd and Peter A. Loeb. An Introduction to Nonstandard Real Analysis. Harcourt Brace Jovanovich, New York, 1985.

[33] Henry Kautz. The logic of persistence. In Proceedings AAAI-86; pages 401–405. AAAI, 1986.

[34] Robert Kowalski. A Logic for Problem Solving. North-Holland, New York, 1979.

[35] Robert Kowalski and M. J. Sergot. A logic-based calculus of events. New Generation Computing, 4:67–95, 1986.

[36] Vladimir Lifschitz. Pointwise circumscription: preliminary report. In Proceedings AAAI-86, pages 406–410. AAAI, 1986.

[37] Vladimir Lifschitz. Formal theories of action. In Proceedings IJCAI 10, pages 966–972. IJCAI, 1987.

[38] *Vladimir Lifschitz and Arkady Rab nov. Miracles in formal theories of action.* Artificial Intelligence, *38:225–237, 1989.*

[39] *David Maier and David Warren.* Computing with Logic: Logic Programming with Prolog. *Addison-Wesley, Reading, Massachusetts, 1988.*

[40] *John McCarthy. Programs with common sense. In Marvin Minsky, editor,* Semantic Information Processing, *pages 403–418. MIT Press, Cambridge, Massachusetts, 1968.*

[41] *John McCarthy. Circumscription - a form of nonmonotonic reasoning.* Artificial Intelligence, *13:295–323, 1980.*

[42] *John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence.* Machine Intelligence, *4:463–502, 1969.*

[43] *Drew V. McDermott. A temporal logic for reasoning about processes and plans.* Cognitive Science, *6:101–155, 1982.*

[44] *Drew V. McDermott and Jon Doyle. Non-monotonic logic i.* Artificial Intelligence, *13:41–72, 1980.*

[45] *L. Morgenstern and Stein L.A. Why things go wrong: A formal theory of causal reasoning. In* Proceedings AAAI-88, *pages 518–523. AAAI, 1988.*

[46] *R. H. Pennington, editor.* Introductory Computer Methods and Numerical Analysis. *Macmillan, 1971.*

[47] *Amir Pnueli. The temporal logic of programs. In* Proceedings of the 18th Annual Symposium on Foundations of Computer Science, *pages 46–57, 1977.*

[48] *V. R. Pratt. Process logic. In* Proceedings of the 6th POPL, *pages 93–100. ACM, 1969.*

[49] *A.N. Prior.* Past, Present, and Future. *Clarendon Press, 1967.*

[50] *A. Ralston and P. Rabinowitz.* A First Course in Numerical Analysis. *McGraw-Hill, New York, 1978.*

[51] Manny Rayner. Did newton solve the "extended prediction problem?". In Brachman et al. [7], pages 381–385.

[52] Raymond Reiter. A logic for default reasoning. Artificial Intelligence, 13:81–132, 1980.

[53] Nicholas Rescher and Alasdair Urquhart. Temporal Logic. Springer-Verlag, New York, 1971.

[54] Erik Sandewall. Combining logic and differential equations for describing real-world systems. In Brachman et al. [7], pages 412–420.

[55] Erik Sandewall. Filter preferential entailment for the logic of action in almost continuous worlds. In Proceedings IJCAI 11. IJCAI, 1989.

[56] L. F. Shampine and M. K. Gordon. Computer Solution of Ordinary Differential Equations. W. H. Freeman and Company, 1975.

[57] Yoav Shoham. Chronological ignorance: Time, nonmonotonicity, necessity, and causal theories. In Proceedings AAAI-86, pages 389–393. AAAI, 1986.

[58] Yoav Shoham. Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence. MIT Press, Cambridge, Massachusetts, 1988.

[59] J. Van Benthem. The Logic of Time. Kluwer Academic Publishers, Boston, Massachusetts, 1983.

[60] Steven Vere. Planning in time: Windows and durations for activities and goals. IEEE Transactions on Pattern Analysis and Machine Intelligence, 5:246–267, 1983.

[61] David E. Wilkins. Domain independent planning: Representation and plan generation. Artificial Intelligence, 22:269–302, 1984.

[62] David E. Wilkins. Practical Planning: Extending the Classical AI Planning Paradigm. Morgan-Kaufmann, Los Altos, California, 1988.

[63] Marianne Winslett. Reasoning about action using a possible worlds approach. In Proceedings AAAI-88, pages 89–93. AAAI, 1988.

Chapter 4

Controlling Processes

This book is concerned with the behavior of processes. The world we live in can be described in terms of a set of interacting processes. In the previous chapter, we discussed how to model the behavior of processes. In this chapter, we begin to consider how to influence that behavior.

✓ two

Some processes are easier to control than others. For instance, someone typing at a word processor generally has a fair amount of control over what characters appear on the screen. Other processes are influenced by a large number of factors only a few of which we are able to directly observe or influence. In sending an electronic mail message, for example, the speed with which the message arrives at its destination is determined in part by the path provided and in part by the traffic on the networks specified in that path. Electronic mail users can directly control the former but have little control over the latter. If you could somehow predict the traffic on the network, then you might be better prepared to specify a path that would speed your message to its destination. Unfortunately, predicting network traffic flow is itself a complicated and time consuming task.

In studying the control of processes, it is often convenient to describe the world in terms of two processes: one of which we have absolute control over, and a second process that we wish to control. The first is called the *controlling process* and the second the *controlled process*. The behavior of the controlling process is determined in part by the control-system designer. Given some desired behavior for the controlled process, the task is to design a device that realizes the controlling process and forces the desired behavior in the controlled process.

✓ plant

°©1990 Thomas Dean. All rights reserved.

The interaction between controlling and controlled processes can be quite complex. We generally think of the controlling process as calling all the shots, but the control exerted by the controlling process over the controlled process is seldom complete. Factors that influence the controlled process but are not under the control of the controlling process have to be accounted for. The controlled process can, and in many cases must, influence the controlling process in order to bring about the desired behavior. This influence is mediated through the use of special devices used by the controlling process to *observe* the behavior of the controlled process.

Information about the observed behavior of the controlled process is often used by the controlling process in determining what action to take next. This basic idea that the responses of the controlling process are computed from the observed behavior of the controlled process is generally referred to as *feedback* control. In some cases, the need for observation can be reduced or even eliminated by using models to *predict* the behavior of the controlled process.

In this chapter, we consider techniques drawn primarily from control theory and control systems engineering. We focus ~~primarily~~ on the role of feedback in the design of control systems with an emphasis on representations and techniques that stress computational issues. We introduce criteria for controllability, observability, stability, and optimality, and consider a variety of problems to illustrate these concepts. We then consider some basic feedback controllers and how they might be embedded in a computational framework. In the context of discussing feedback control, we introduce programming approaches that are well suited to building control systems that have to be particularly responsive to change. We end this chapter by considering a problem in robotics that lies at the boundary between those problems traditionally considered within the purview of control theory and problems associated with artificial intelligence. The objective here is not to provide a comprehensive survey of control techniques, but rather to draw on the control disciplines for insights and general techniques that apply to the full range of planning and control problems. Before launching into the more technical discussions drawing on results from control theory, we consider a particular problem to illustrate some basic issues.

4.1 Robot Navigation as a Control Problem

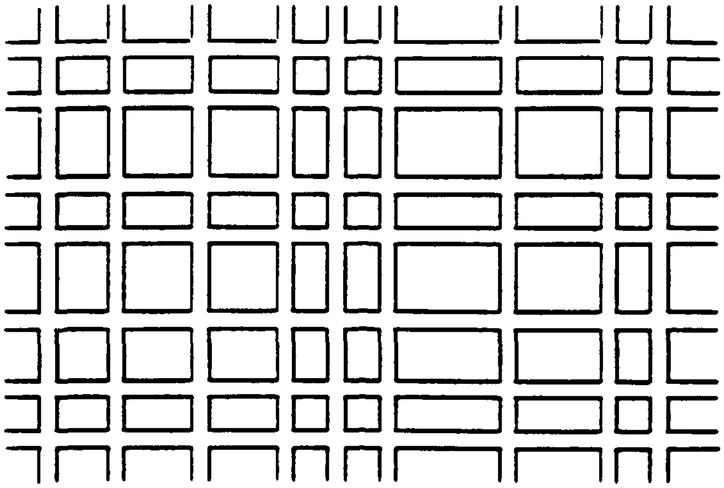


Figure 4.1: A city street layout

Consider the following control problem. Suppose that you want to control a robot to move from one location to another in a city. The robot has to travel using city streets that are arranged as an irregularly-spaced grid of two-way streets (see Figure 4.1). You have to devise a control algorithm to direct the robot to move from its present location to a destination location defined in terms of global coordinates. Of course, the problem is not yet well enough specified that you can run off and start writing down an algorithm. There are a number of other factors that we have to consider.

First, what sort of control can we exert over the robot? Most likely there will be some means of controlling the robot's speed and direction of travel, but it's not likely that the robot will move exactly where we tell it nor will it move at precisely the speed that we specify. If we indicate that the robot is to move due South at 12 kilometers per hour and there is a brick wall in the way, then we might expect some difference between the specified and the actual speed and heading. Usually, however, the differences between actual and specified control variables are more subtle. Errors accumulate and combine in executing a sequence of control actions. Sooner or later it becomes necessary to compare the actual effect against the intended effect, and this is where sensors enter into the picture.

Sensors are used to monitor the progress of the robot and to determine the state of the environment. Sensors can determine and correct for movement error. For instance, the robot might be equipped with shaft encoders for determining how many revolutions the drive wheels have turned or what direction the wheels are pointing. From this information. we can compute an estimate of where the robot is relative to where it started out. Sensors and the estimates derived from sensor data are also subject to errors. Somehow or another we have to take such errors into account. For instance. it may be that the errors are known to satisfy a particular statistical distribution from which we can calculate a measure of how certain we are in the inferences derived from sensor data. If our confidence in our inferences is low. then that could mean that we lack sufficient information to formulate a good answer to the control problem we are faced with. In some cases. being left with insufficient information is unavoidable and we must proceed to schedule critical control actions with whatever information we have at hand. In other cases. we can use sensors to gather additional information so as to make inferences that we are more confident in.

Sensors tell us about more than just the state of the robot: they tell us about the state of the larger world in which the robot is embedded. In the simplest robot navigation tasks. the only thing that changes is the robot itself and its position in the world. The environment is said to be static. If we know something about the fixed state of the environment, then we can take advantage of this in designing a control algorithm. Knowledge of the environment might take the form of a map labeled with street names. whether or not traffic moves in one direction or both, and whether there are stop signs or other impediments to traffic flow.

In more realistic problems. the environment changes; there are other vehicles on the road, traffic lights change, roads are blocked by construction. and pedestrians occasionally dart out into traffic. The static map may still be useful, but often we can supplement our knowledge of the environment to account for dynamic phenomena. For instance, we might have access to a construction schedule indicating where and when certain streets will be closed to traffic. In some cases, we might be able to model certain disturbances as predictable processes. A construction crew might be laying new gas pipe under a particular street at the rate of one block per night so that at most one block-long section of the street is impassable on any given night. If you notice the crew laying pipe on any two nights, you can predict what block will be closed off for any subsequent night.

While some processes are predictable. others are either difficult to predict (*e.g.*, jay-walking pedestrians) or not worth the trouble (*e.g.*, traffic lights). In order to deal with such processes, the control algorithm has to be alert to changes in the environment that indicate the existence of processes whose behavior might have an impact on the performance of the robot. The robot has to be continually alert for evidence of certain processes (*e.g.*, pedestrians straying into the street in front of the robot). Other processes need only be monitored in certain circumstances. For instance, the robot has to check for the state of the traffic light at the next intersection only as it approaches that intersection. The design of the control algorithm must take into account the sensors available and the tasks they are to be put to. Sensors often constitute a scarce resource in need of careful management.

There is another aspect of the control of our mobile robot that we have carefully avoided up until now, and that concerns how the algorithm that we devise is to be implemented. In order to implement a control algorithm, we need to specify the algorithm in terms of a language, and we have to provide a compiler for that language, and a target machine for the code generated by the compiler. In fact, it generally is difficult to specify a control algorithm without some specific implementation in mind.

How long a series of program statements takes to execute on a particular machine may be critical in determining the consequences of a control action. For instance, suppose that you want to compute how to respond in the case in which a pedestrian runs out into the street in front of the robot. Certainly it would be a good idea to apply the brakes as soon as possible if indeed that is an appropriate thing to do. How long the algorithm takes to compute whether or not to apply the brakes will have a profound impact on the health of the pedestrian in question. If the robot is to swerve in an attempt to avoid hitting the pedestrian, then the direction in which the wheels are turned will depend upon the time that they are turned, and this will depend upon the time it takes to compute the direction.

In some cases, we can just assume that the time required to compute responses is shorter than the time available for computation. For instance, suppose that at time t the robot interprets its sensor data as indicating a pedestrian standing in the street 5 meters directly in front of it. The robot attempts to compute what action to take at time $t + \Delta$. The control algorithm is implemented so that the time required to compute such a response is less than Δ. Having computed an appropriate answer, the control algorithm might simply wait out the remaining time, or hand the action and the time it is to be executed .o a sequencer responsible for executing actions at specified times. Of course, if the robot is traveling at a meter a second and Δ is longer than a couple of seconds, then the response will likely be too late to be of any use.

Some of the decisions concerning how long to spend computing an appro-

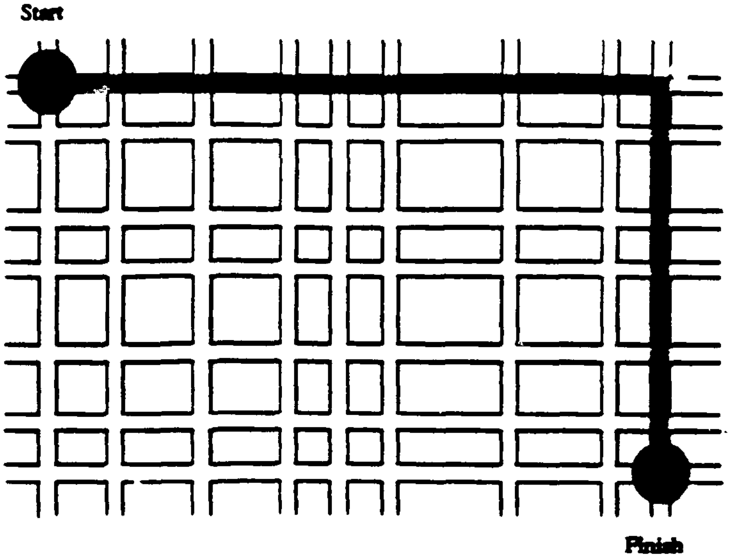


Figure 4.2: A path generated by dead reckoning

priate response in a given set of circumstances can be carried out at design time. Other decisions concerning how to long to compute are better left until run time when the allocation of computational resources can be based on more data about the situation at hand. If the lead time for responding to a certain sort of phenomena varies, then having a rigid scheme for computing a response may lead to poor performance on average. Jamming on the brakes is only appropriate as a last resort. In situations where more time is available to arrive at a decision, a more careful analysis is often called for. In this chapter, we ignore many of the issues that relate to the run-time allocation of processor time to optimize decision making. Chapter 8 directly addresses these issues. In this chapter, we take a conservative approach to ensure that the algorithms that we develop perform reasonably for even the worst-case situations anticipated.

So far we have considered several factors that are important in specifying control problems. Now, we consider some specific control problems. In an ideal world, when the robot is told to turn left 15° and move forward at 2 meters per second for 5 seconds, the robot ends up exactly 10 meters from its original position facing 15° counter clockwise from its original heading. Consider the problem involving a static environment in which all of the streets allow two-way traffic and are obstacle free and the robot is standing in the center of an intersection and is instructed to move to the center of a second intersection specified in x and y coordinates in the frame of reference of the robot's initial position. In this case, an appropriate control algorithm would direct the robot to complete the traversal in two steps following the paths indicated by the x and y offsets (see Figure 4.2).

In the above ideal world, the robot is said to direct itself by "dead reckoning." Aside from a clock to measure the passage of time, and thereby gauge the distance traveled, the robot requires no sensors to direct its motion. Suppose that we relax the requirement that the robot be able to control its velocity precisely. In this case, it is possible that the robot's estimates of distance traveled are subject to error. How is the problem changed as a consequence? If the errors are small relative to the length of a city block, a simple variation on the dead-reckoning approach will work just fine. If the errors are large, then the problem may be impossible to solve since the robot will have no way to determine if it reaches its destination. Even if the robot has some other means of detecting that it has arrived at its sought-after destination, significant movement errors may force the control algorithm to randomly choose paths.

Suppose that the robot can determine its position at any time in some global coordinate system. Now movement errors can be corrected by what is generally referred to as *feedback*. The control algorithm attempts to move 5 meters to the left; it checks to see how far it actually moved: it attempts to correct for the error observed. As long as the errors are some fraction of the distance attempted, this technique will converge quickly on the desired distance. If determining global position is fast enough, then this technique reduces to the previous dead-reckoning method.

Now suppose that all streets are not passable; some streets are one way and others are blocked by construction equipment. The dead-reckoning approach will obviously not work, but a simple path-following strategy will suffice to find a path if one exists. Figure 4.3 shows the streets traversed by the robot under the control of a simple path-following algorithm that tries to shorten the Euclidean distance to the destination whenever possible, backing up only when its way becomes blocked. The problem is that directing the robot using the simple path-finding strategy causes the robot to traverse streets that it might not have if it possessed a more global perspective of the city.

Suppose that the robot has an accurate map of the city indicating one-way streets and construction road blocks. Rather than actually traversing

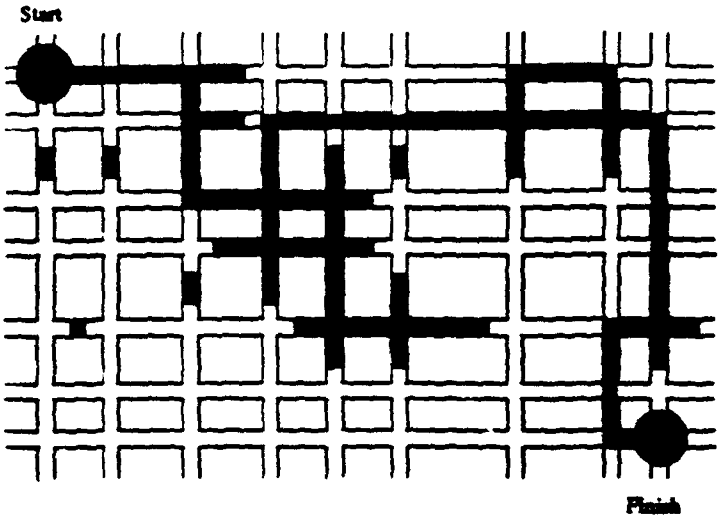


Figure 4.3: Navigation without the aid of a map

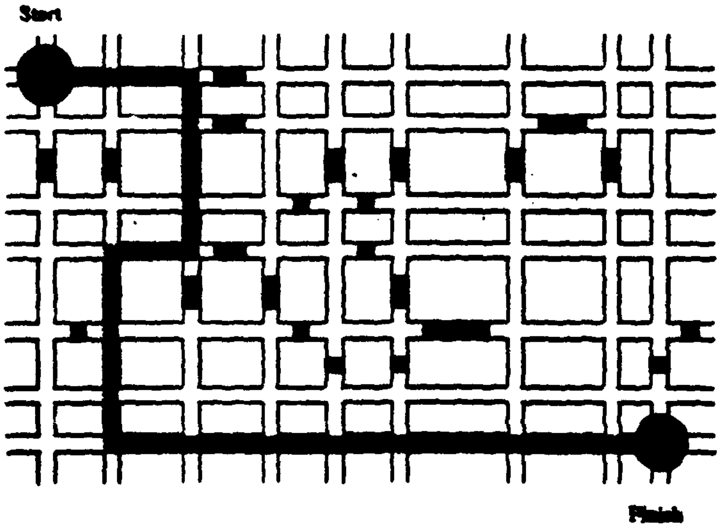


Figure 4.4: Navigation using path planning and a global map

the streets, the control algorithm could use the map to *simulate* traversing the streets and thereby find a short path. Computing the shortest path between any two locations can be done in $O(n^2 \log n)$ time using Dijkstra's algorithm [1], assuming a square grid of streets with n streets along each axis of the grid. Figure 4.4 shows the streets traversed by the robot under the control an algorithm with access to a map. This method of simulating the behavior of the robot in order to eliminate unnecessary work or avoid an undesirable effect represents an instance of *feedforward*. The control algorithm generates and analyzes possible actions and their consequences so that it can choose among the available options.

√

best first search

The use of feedback and feedforward are common in the design of control systems. Feedback compensates for a system's inability to accurately predict the effects of a control action on the behavior of a controlled process. Feedback relies on being able to accurately monitor the behavior of a process. Feedforward enables a system to anticipate both desirable and undesirable consequences and take steps to, respectively, take advantage of or avoid them. Feedforward relies on a system having an accurate model for the process being controlled.

Feedforward and feedback complement one another. In situations in which the controlled process cannot be accurately predicted but can be closely monitored, tight feedback loops enable a control algorithm to generate control actions on the basis of immediately past performance. Such a scheme is likely to work assuming that the factors influencing the process at one point in time are similar in type and magnitude to the factors influencing the process a short time previously. In situations in which the controlled process cannot be accurately monitored but can be accurately predicted, control actions are generated in response to predictions concerning the processes behavior. If the process can't be monitored at all, then control proceeds blindly relying on the accuracy of the predictive model.

Traditional methods in planning stress the use of feedforward methods whereas traditional methods in control stress the use of feedback. The reason for their different emphases is easy to explain. First of all, planning is by definition concerned with predicting the future in order to guide behavior. Much of the early work in planning was concerned with processes that interact with one another in a complex manner, and, hence, influencing the behavior of these processes required anticipating these interactions. This early work generally assumed that the controlled process, while complex, was understood well enough to be accurately modeled. More recent work has begun to relax this assumption by either using feedback to supplement predictions or using stochastic models that take uncertainty into account.

In contrast with the work in planning, much of the early work in control assumed that the controlled process was subject to a multitude of factors that either were not well understood or required run-time data that simply was not available. Precise adjustments to the control parameters were needed to achieve the desired behavior requiring that the controlling process be able to generate the necessary control actions at a high rate. A more complex algorithm for determining the next control action lowers the rate at which control actions can be generated, whereas, the more inaccurate the models are in predicting the effect of control actions, the more frequently the controlling process has to be monitored and the control parameters adjusted to compensate for the inaccuracies of the model. In the past, many industrial control applications have favored trading model complexity for increased reliance on feedback and higher parameter-adjustment rates. As computers become faster and our modeling techniques more reliable, there has been a tendency to incorporate more and more complex modeling techniques into industrial controllers. If this trend continues, industrial controllers will begin to look more like planners.

As the control community begins to realize the advantages of increased computational power for supporting complex modeling, so the planning community is beginning to realize the problems in relying solely on the predictions of a complex model. Correcting these problems is not simply a matter of building an interpreter that executes a sequence of actions generated by a traditional planner and occasionally senses the environment to see if the actions have had their desired effect. The problem with this approach is that the controlled and the controlling processes are often out of synch with one another.

A control action generated one moment may be deemed inappropriate at the next as new information becomes available. To simply generate a sequence of actions and expect that the sequence can be carried out without modification is for many problems absurd. In asking directions in Boston, a local may tell you to turn left on Commonwealth Avenue and follow it for three blocks until you get to Massachusetts, but if you find four fire trucks tying up traffic on Commonwealth Avenue, then you would be well advised to disregard their directions and find an alternative route. There was nothing wrong with the directions provided given what was known at the time they were solicited, but knowledge changes over time and such changes should be taken into account when deciding how to act.

Avenue

Of course, the preceding paragraph shouldn't be taken as an argument against planning; we've already seen that path planning can lead to improved performance in certain circumstances. What we have to beware of is blindly executing plans in the face of information that warns against their use. The traditional notion of a plan as a sequence of actions has to be rethought. Plans should be interpreted as suggestions about how to behave. Some suggestions require a long time to generate. but the processes that they are designed to help control may proceed at a similarly slow pace. In real-world problems. there are any number of processes that require some amount of control. Some processes proceed slowly and require attention only at widely-spaced intervals (*e.g.*. the pipe-laying process discussed earlier). Other processes are faster paced and require almost constant attention (*e.g.*. pedestrian traffic). The trick is to deal effectively with the fast-paced processes (*e.g.*. steer clear of pedestrians and stop at appropriate traffic signals) while at the same time directing behavior so as to take into account suggestions regarding the slower processes (*e.g.*. avoid routes that are believed to be obstructed by construction) and suggestions generated off-line as it were regarding faster-paced processes (*e.g.*. if you see a ball rolling out into the street. brake hard as a child may be following closely behind).

In the following, it will be useful to separate out two kinds of control algorithm. One that generates suggestions concerning certain low-level behaviors and that is likely to perform out of synch with the processes whose behavior it is meant to influence. and a second that is closely tied to the processes that it is meant to influence. The distinction is artificial: it serves primarily to identify two distinct mind sets that have to be merged in order to develop a coherent theory of control. To provide a label for the two kinds of control and identify the source for the corresponding mind sets, we call the first *high-level planning* and the second *low-level control*. An example of a high-level planning algorithm would be a path planning algorithm designed to influence the movement of the robot. An example of a low-level control algorithm would be the algorithm that directs the speed and heading of the robot as it traverses the city streets avoiding obstacles and maneuvering around corners.

One possible architecture for a system integrating high-level planning and low-level control might consist of two components: a reactive component that determines what to do at the next instant. and a strategic component that attempts to mediate the behavior of the reactive component by imposing constraints on the behavior of the low-level systems. It is up to the low-level system to interpret these constraints so as to adjust its behavior while at the same time maintaining real-time performance.

In this chapter, we are primarily interested in what we have called low-level control. Toward the end of this chapter, however, we begin to address high-level control issues as prologue to the next chapter which will deal almost exclusively with high-level strategic planning. Now, we draw upon the disciplines of control theory and control systems engineering to develop some terminology and explore techiques that will be used in subsequent chapters.

4.2 Controllability

Consider the following time-invariant discrete-time dynamical system.

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) \\ y(k) &= g(x(k)) \end{aligned}$$

The state transition function, f, completely determines the state of the system at time $k+1$ given the state and the input at time k. Initially, we assume that the state of the system is directly observable, and so the output function, g, is defined

$$g(x(k)) = x(k).$$

In solving a particular control problem, we are interested in generating appropriate inputs so as to constrain the behavior of the dynamical system. In Chapter 1, we introduced a general formulation of the control problem, representing the behavior of a dynamical system in terms of the set of possible state-space trajectories.

$$H_X \triangleq \{h_X : T \rightarrow X\}.$$

In this formulation of the problem, the desired behavior of the system is specified in terms of a *goal* set.

$$G \subset H_X.$$

There are several special cases of this formulation that we consider in the following sections.

In the *servo problem*, we are given a *reference trajectory*, and expected to repeat or *track* that trajectory as closely as possible. In the *set-point regulation* problem, the objective is for the system to achieve and maintain a particular state or set of states starting from any initial state. In the terminology of Chapter 2, we wish to find some input function $v \in \{v : T \rightarrow U\}$ so that for any initial time $\tau \in T$ and initial state $x(\tau) \in X$ there exists $t > \tau$ such that for all $t' > t$ we have

$$f(x(t'), v(t')) \in C,$$

where $C \subset X$ is the set of target states.

We can generalize on our formulation of the set-point regulation problem to restrict not only the final states of the system, but the intermediate states as well, thereby restricting the motions (state space trajectories) of the system. For instance, we might require that the system avoid a certain set of states, by stipulating that for all $t > \tau$ we have

$$f(x(t), v(t)) \notin Q,$$

where $Q \subset X$ is the set of states to avoid and $C \cap Q = \emptyset$.

Among the qualitative properties of dynamical systems and their controllers, the following notion of *controllability* is particularly relevant to the set-point regulation problem. An event $\langle \tau, x \rangle$ in the phase space defined by $T \times X$ is said to be *controllable with respect to a set of target states*, $C \subset X$, if and only if there is some time t and some input v which moves $\langle \tau, x \rangle$ into the set $\{t : t \geq \tau\} \times C$. A dynamical system is *completely controllable with respect to* C if and only if every event in $T \times X$ is controllable with respect to C. This notion of complete controllability with respect to a set of target states provides necessary and sufficient conditions for there being a solution to the set-point regulation problem.

As was mentioned in Chapter 2, one of the best developed areas of modern control theory concerns the analysis of dynamical systems that can be modeled as linear multivariable systems. In this chapter, we illustrate the power of linear systems theory by defining three important qualitative properties of dynamical systems, and stating simple mathematical criteria for these properties to be satisfied.

We begin with the notion of controllability. Criteria for controllability are generally specific to a particular method of modeling dynamical systems. In general, we are interested in whether or not it is possible to transfer any state $x(t_0) \in X$ to any other state in X in a finite amount of time $t_1 - t_0$ where $t_0 < t_1$ by appropriately choosing $u(t)$ for $t_0 \leq t \leq t_1$. If such arbitrary transfers are possible, we say that the system is *completely controllable* (no restriction to a particular set of target states).

Consider the following linear time-invariant system represented by

$$\begin{aligned}\dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(t)\\ \mathbf{y}(t) &= C\mathbf{x}(t)\end{aligned}$$

where $\mathbf{x}$ is the n-dimensional state vector, $\mathbf{u}$ is the p-dimensional input vector, $\mathbf{y}$ is the q-dimensional output vector, and A, B, and C are, respectively, $n \times n$, $n \times p$, and $q \times n$ real constant matrices. There are a number of relatively simple mathematical conditions for such a system being completely controllable. One of the simplest is provided by the following theorem which is stated here without proof (see Chen [9] or Gopal [14] for proofs and related theorems).

Theorem 1 *The system is completely controllable if and only if the rank*[1] *of the* $n \times np$ controllability matrix, $[B|AB|\cdots|A^{n-1}B]$, *is* n.

As a simple example, the dynamical system for the single-degree-of-freedom robot introduced in Chapter 2 with state equation,

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 1/M \end{bmatrix} \mathbf{u}(t).$$

is completely controllable since the rank of its controllability matrix,

$$[B|AB] = \begin{bmatrix} 0 & 1/M \\ 1/M & 0 \end{bmatrix},$$

is 2. However, the system described by

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & C_1 \\ C_2 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \mathbf{u}(t),$$

has a controllability matrix,

$$[B|AB] = \begin{bmatrix} 1 & C_1 \\ 1 & C_2 \end{bmatrix}.$$

[1] The *rank* of an $n \times m$ rectangular matrix, A, is defined as the maximum number of linearly independent column vectors, or, equivalently, the order of the largest square array whose determinant is non-zero, where the square array is obtained by removing rows and columns from A.

indicating that the system is controllable only if $C_1 \neq C_2$.

There are other similarly concise and equivalent conditions stated in the literature. Both Chen [9] and Gopal [14] provide similar results for linear time-varying systems, as well as constructive proofs that identify the appropriate input functions. It is testimony to the power of linear systems theory that such precise conditions can be stated for such a general class of dynamical systems.[2]

It should be noted that the above stated notion of controllability places no constraint on the input (controller) or on the trajectory followed by the system. A system may be determined as uncontrollable by the above criterion, while being controllable in most practical respects. For instance, the system may move to any given state from all initial states that will arise in practice. As another example, we may not care about certain components of the state vector: it may be that we are only concerned with controlling the output of the system.

To investigate further the notion of controllability, we consider some examples of dynamical systems that can be represented in terms of finite state automata. These dynamical systems are referred to as *discrete event systems* in the literature [25]. We represent a discrete event system as an automaton, $G = (U, X, f, x_0)$, where, in keeping with our previous notation, U is the set of inputs (think of U as a set of primitive events), X is the set of states, $f : U \times X \rightarrow X$ is the state transition function, and x_0 is the initial state.

We partition U into two sets: U_c, the set of *controllable* events, and U_u, the set of *uncontrollable* events. An *admissible control* for such a dynamical system consists of a subset $\gamma \subseteq U$ such that $U_u \subseteq \gamma$. Let $\Gamma \subseteq 2^U$ represent the set of all admissible controls. If $\gamma \in \Gamma$ and $u \in \gamma$, we say that u is *enabled* by γ, otherwise we say that it is *disabled*. A controller for a given dynamical system is specified as a map

$$\eta : X \rightarrow \Gamma.$$

The idea is that disabled events are prevented from occurring and enabled events are allowed to occur if permitted by the underlying dynamics. The

[2] As was noted in Chapter 2, it is standard practice in engineering control systems to model real-world nonlinear systems using linear approximations. Since small perturbations of the elements of the matrices A and B may signal the difference between controllability and its lack, it should be noted that statements of system controllability must be carefully weighed in the process of design.

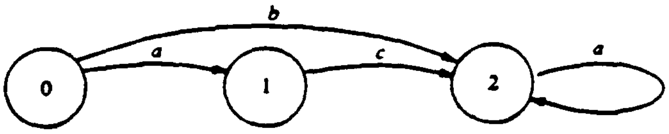


Figure 4.5: A dynamical system represented as a finite state automaton

stipulation that $U_u \subseteq \gamma$ for all $\gamma \in \Gamma$ captures the intuition that the controller cannot prevent the uncontrolled events from occurring if the dynamics dictates otherwise. An issue arises regarding what happens if all of the events for a given state are disabled. We resolve the issue by simply requiring that the controller ensure that for any state there is at least one enabled event for which the transition function is defined: the system can remain in the same state only if that is permitted by the dynamics.

Consider the dynamical system depicted in Figure 4.5 in which $U = \{a, b, c\}$, $X = \{0, 1, 2\}$, $x_0 = 0$, and f is defined so that

$$(0, a) \mapsto 1, (0, b) \mapsto 2, (1, c) \mapsto 2, \text{ and } (2, a) \mapsto 2.$$

Let $U_c = \{a, b\}$ and suppose that we wish to design a controller that achieves $\{2\}$ while avoiding $\{1\}$. The controller defined by

$$0 \mapsto \{b\} \text{ and } 2 \mapsto \{a\}$$

will suffice to do exactly what we want. The same controller will work if $U_c = \{a\}$. However, if we have $U_c = \{b\}$, then there is no controller satisfying the requirements given.

There is an alternative approach to characterizing the behavior of discrete event systems modeled as finite state automata. In formal language theory, a finite state automaton can be viewed as a generator for a language. Let U^* denote the set of all finite strings of elements of the set U. A subset $L \subseteq U^*$ is called a *language* over U. The automaton described above is a generator for the language

$$L = ba^* + aca^*,$$

indicating the union of the set of strings consisting of b followed by a finite number of a's, and the set of strings consisting of a followed by c followed by a finite number of a's. Instead of asking if we can design a controller that achieves $\{2\}$ while avoiding $\{1\}$, we ask if we can design a controller for the automaton so that it generates the language $L' = ba^* \subseteq L$.

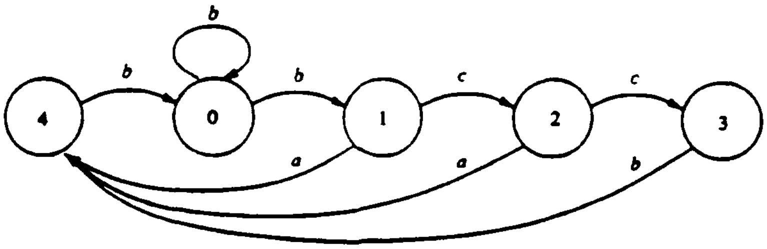


Figure 4.6: One component of a product system

Ramadge and Wonham [25] define a *supervisor* for a discrete event system as a map

$$\eta : L - \Gamma.$$

where L is the language (or *behavior*) generated by the discrete event system. The *prefix closure* of $L \subseteq U^*$ is that subset $L \subseteq U^*$ defined by

$$\bar{L} = \{u : uv \in L \text{ for some } v \in U^*\}.$$

A language $K \subseteq L$ is said to be *controllable* with respect to a given discrete event system if

$$\bar{K}U_u \cap L \subseteq \bar{K},$$

where $\bar{K}U_u$ represents the set of all strings consisting of a string from the prefix closure of K concatenated with an event from U_u. In [25], Ramadge and Wonham prove the following, thus providing necessary and sufficient conditions for the existence of supervisors for discrete event systems.

Theorem 2 *For any discrete event system A with closed behavior L and any subset $K \subseteq L$, there exists a supervisor that serves to restrict A to exactly K if and only if $\bar{K} = K$ and K is controllable.*

In some cases, it is convenient to represent a dynamical system as a collection of finite state automata loosely coupled through the state space resulting from taking the cross product of the state spaces for the individual automata. As an example, suppose that we wish to model a collection of n identical chemical processes. Each individual process is modeled by an automaton $G_i = (U_i, X_i, f_i, x_{0_i})$ where the ith automaton is defined by $U_i = \{a_i, b_i, c_i\}$, $X_i = \{0_i, 1_i, 2_i, 3_i, 4_i\}$, $x_{0_i} = 0$, and f_i is as indicated in Figure 4.6. Let $U_{c_i} = \{a_i, c_i\}$. Suppose that all n processes run independently of one another with one important exception: state 4 involves the use of a piece of equipment with limited capacity such that only one process can be in state 4 at a time. We wish to design a controller that will guarantee this. Note that once a process enters state 1, we can exercise some control over *when* it enters State 4, but we can only delay this event, we cannot prevent it from happening.

To represent the combined behavior of the collection of processes, we define the *product generator* $G = \{U, X, f, x_0\}$ where $U = \cup_{i=1}^{n} U_i$, $X \triangleq \Pi_{i=1}^{n} X_i$, $U_c = \cup_{i=1}^{n} U_{c_i}$, $x_0 = (x_{0_1}, x_{0_2}, \ldots, x_{0_n})$ and for each $u \in U_i$ we have

$$f(u, (x_1, x_2, \ldots, x_i, \ldots, x_n)) = (x_1, x_2, \ldots, f_i(u, x_i), \ldots, x_n)).$$

The objective is to build a controller for G such that at most one of the chemical processes is in the state requiring the piece of equipment at any given point in time.

In the worst case, all of the processes will simultaneously arrive at state 1 in their respective state spaces. At this point, exactly one process can transition to state 4, while the $n - 1$ remaining processes are forced to enter state 2. The same simple analysis applied to state 1 can be applied to state 2 with the conclusion that $n - 2$ processes are forced to enter state 3. The controller has no control over the processes in state 3, and hence we conclude that there exists a controller for the product system if and only if $n \leq 3$.

Discrete event systems can be used to model manufacturing systems, communication networks, vehicular traffic problems, and a variety of other dynamical systems requiring coordination and control. In addition to answering mathematical questions concerning the existence of supervisors, the current theory provides constructive methods for realizing certain classes of supervisors. In the best circumstances, these methods require time and storage polynomial in the size of the state space. For practical problems, one generally has to be clever in searching the space of possible controllers for one that satisfies the domain constraints.

4.3 Observability

So far, we have had little to say about the role of the system output function. In fact, we initially assumed that $y(t) = g(x(t)) = x(t)$, so that the state of the system was directly observable as output. In general, the entire system state will not be directly observable. If the controller requires either the entire system state vector or specific components of this vector. then an additional module has to be added to the control system in order to recover the state by observing the system output. Such modules are generally referred as *observers*. If the function g is known and invertible. then the construction of an observer is trivial. Generally. g is not invertible and the state has to be recovered by observing the output of the system over some interval of time. In the following, we consider a notion of observability which. at least in the case of linear multivariable systems. turns out to be closely related to controllability.

A system is said to be *completely observable* if it is possible to identify any state $x(t_0) \in X$ by observing the output $y(t)$ for $t_0 \leq t \leq t_1$ where $t_0 < t_1$. *observation problem.* The problem stated is traditionally called *the* observation problem. but it is actually just one of several so-called *state-determination* problems. The observation problem involves determining the state from future outputs. There is a related problem called the *reconstruction problem* that involves identifying the state from past outputs: identify the state $x(t_1) \in X$ by observing the output $y(t)$ for $t_0 \leq t \leq t_1$ where $t_0 < t_1$. As in the case of controllability. there are simple mathematical criteria for observability in linear multivariable systems (see Chen [9] or Gopal [14] for proofs and equivalent conditions).

Theorem 3 *The system is completely observable if and only if the rank of the $nq \times n$* observability matrix.

$$\begin{bmatrix} C \\ CA \\ \vdots \\ CA^n - 1 \end{bmatrix} \text{ is } n.$$

Given the similarity of the statement of Theorems 1 and 3 one might suspect that there is a rather deep relationship between controllability and observability for linear multivariable systems. It would be particularly convenient if one could prove that a system is observable if and only if it is controllable. This happens to be true in a somewhat convoluted mathematical sense as we see in the following theorem.

Theorem 4 *(The Principle of Duality) The system represented by*

$$\begin{aligned} \dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) \end{aligned}$$

is controllable (observable) at time t_0 if and only if the dual system represented by

$$\begin{aligned} \dot{\mathbf{z}}(t) &= -A'\mathbf{z}(t) + C''\mathbf{v}(t) \\ \mathbf{w}(t) &= B'\mathbf{z}(t) \end{aligned}$$

is observable (controllable) at t_0, where the prime (e.g., B') indicates matrix transposition, and the second system (called the adjoint*) is mathematically closely related to the first.*

One practical consequence of Theorem 4 is that once you have constructed a controller (observer), you have done all the necessary work required to construct the associated observer (controller): the algorithms required for one task are almost identical to the algorithms required for the other task. It is also interesting to note that observability and controllability in linear systems can be considered independently. The two problems of building a controller and building an observer can be pursued independently of one another. The two problems are said to be *separable*. This separation property does not hold in general.

Results similar to that of Theorem 4 hold for linear systems corrupted with Gaussian noise. In Chapter 6, we consider the problem of building a deterministic regulator (controller) and a stochastic estimator (observer) for dynamical systems modeled as linear systems corrupted with Gaussian noise. It turns out that these two problems are also separable: by coupling the optimal deterministic regulator to the optimal stochastic estimator one has constructed an optimal control system.

It should be emphasized that the notion of observability introduced in this section is quite strong. In general, a controller need not reconstruct the entire system state in order to provide satisfactory performance for a given control problem. In many cases, the task of reconstructing the entire system state would impose a significant computational burden. Practically speaking, we are interested in *demand-driven observation strategies* that allocate resources to measurement and interpretation in keeping with the immediate demands on the system. The task-based planning methods presented in Chapter 5 employ this sort of demand-driven observation strategies.

4.4 Stability

When we first introduced the notion of controllability in Section 4.2, we were interested in the ability to first achieve a given state or set of states in a finite amount of time, and then maintain the system in that state of set of states for all time hence. When we subsequently considered controllability criteria for linear systems, we dropped the latter requirement. In many applications, however, it is not enough for a controller to simply move the system to a particular state. Neither is it reasonable to expect that the controller maintain a given state in the face of arbitrary disturbances or perturbations of the dynamical system. Stability is a property of dynamical systems which implies that small changes in input or initial conditions do not result in large changes in system behavior. Stability is not a prerequisite for being able to control a system, but it makes the task of designing a control system somewhat easier. The system describing the inverted pendulum presented in Chapter 2 is not stable by the criteria that we will present shortly, but it is controllable. The concept of stability introduced in the following is attributed to the Russian mathematician A. M. Lyapunov.

We will be concerned with the same linear multivariable system introduced earlier.

$$\begin{aligned} \dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(i) \\ \mathbf{y}(t) &= C\mathbf{x}(t) \end{aligned}$$

Let $\mathbf{u}(t) = \mathbf{u}_c$ be any constant input. If there exists a point $x_e \in \mathbf{R}^n$ such that

$$A\mathbf{x}_e + B\mathbf{u}_c = 0,$$

then $\mathbf{x}_e$ is said to be an *equilibrium point* of the system corresponding to the input $\mathbf{u}_c$. We assume that the system has only one equilibrium point, and, without loss of generality, take the origin of the state space to be that equilibrium point. Finally, we consider only the case in which $\mathbf{u}_c = 0$ so that

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t).$$

This system is *stable in the sense of Lyapunov* at the origin if, for every $\epsilon > 0$, there exists $\delta > 0$ such that $\|\mathbf{x}(t_0)\| \leq \delta$ implies $\|\mathbf{x}(t)\| \leq \epsilon$ for all $t \geq t_0$, where $\|\mathbf{x}\|$ denotes the Euclidean norm for a vector $\mathbf{x}$ of n components $x_1, x_2, \ldots, x_n$ defined by

$$\|\mathbf{x}\| = (x_1^2, x_2^2, \ldots, x_n^2)^{1/2}.$$

The hyper-spherical region defined by the set of all points such that $\|\mathbf{x}\| \leq \epsilon$ serves to ensure a bound on the system response.

We say that the above system is *asymptotically stable* at the origin if

1. it is stable in the sense of Lyapunov. and
2. there exists a real number $r > 0$ such that

$$\|\mathbf{x}(t_0)\| \leq r \text{ implies } \mathbf{x}(t) \rightarrow 0 \text{ as } t \rightarrow \infty.$$

The stability of a linear multivariable system can be determined using a relatively simple mathematical test provided in the following theorem (see [14] for proof).

Theorem 5 *The system described by the state equation,*

$$\dot{\mathbf{x}} = A\mathbf{x}(t) + B\mathbf{u}(t),$$

is asymptotically stable if and only if all of the eigenvalues of the matrix A have negative real parts.

Recall that the eigenvalues of a matrix A correspond to those values of λ such that $\mathrm{Det}(\lambda I - A) = 0$. where I is the identity matrix and $\mathrm{Det}(M)$ indicates the determinant of the matrix M. One particularly convenient advantage of the stability test introduced in Theorem 5 is that it does not require one to solve the system state equations. In the case of the single-degree-of-freedom robot. the eigenvalues correspond to solutions of

$$\mathrm{Det}\left(\begin{bmatrix} \lambda & -1 \\ 0 & \lambda \end{bmatrix}\right) = \lambda^2 = 0.$$

The equation $\lambda^2 = 0$ is called the *characteristic* equation, and, in this case. the characteristic equation has no solutions indicating that the the dynamical system for the single-degree-of-freedom robot is stable.

In the case of the inverted pendulum example of Chapter 2.

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5809 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 4.4537 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 0.9211 \\ 0 \\ -0.3947 \end{bmatrix} \mathbf{u}(t),$$

the characteristic equation is

$$\mathrm{Det}\left(\begin{bmatrix} \lambda & 1 & 0 & 0 \\ 0 & \lambda & -0.5809 & 0 \\ 0 & 0 & \lambda & 1 \\ 0 & 0 & 4.4537 & \lambda \end{bmatrix}\right) = \lambda(\lambda(\lambda^2 - 4.4537)) = 0.$$

$A = I$

Pg 329 Ex ~~5.9~~ 9.4
stabilize

According to criterion established in Theorem 5, the dynamical system for the inverted pendulum is not stable since one of the solutions of the characteristic equation is $\lambda = +\sqrt{4.4537}$.

Before we leave the subject of stability, it is worth mentioning one particularly useful technique referred to as the *root-locus method* developed by W.R. Evans for investigating the stability of linear systems. The root-locus method is most closely associated with what is called classical control theory which, as was mentioned in Chapter 2, is based primarily upon the use of the Laplace transform and analysis in the frequency domain.

Many control systems have a single input variable and a single output variable. The input is referred to as a *reference signal* indicating the desired value for the output or controlled variable. The *transfer function* of such a control system is defined to be the ratio of the Laplace transform of the input variable to the Laplace transform of the output variable. Consider the spring-mass-dashpot system described in Chapter 2, and suppose that we allow an external force to act on the block. The equation of motion of the block is

$$M\frac{d^2x}{dt^2} + C\frac{dx}{dt} + Kx = u(t)$$

~~(4.1)~~

where the output of the system is defined to be x and the input is u. The Laplace transform of Equation 4.1 is

$$Ms^2X(s) + CsX(s) + KX(s) = U(s)$$

~~(3.2)~~

assuming the initial conditions

$$x(0) = x_0, \qquad \frac{dx(0)}{dt} = 0.$$

The transfer function for the system corresponding to Equation 4.2 is

$$T(s) = \frac{X(s)}{U(s)} = \frac{1}{Ms^2 + Cs + K}.$$

By analyzing the system's *poles* (the roots of the denominator or *characteristic equation* of the transfer function) and *zeros* (the roots of the numerator of the transfer function), one can tell a great deal about the transient response characteristics of the control system. For instance, it is well known [10] that, for a system to be ^ stable, it is necessary and sufficient that all of the poles of the system transfer function have negative real parts.[3]

Asymptotically.

[3] The Laplace variable is a complex variable and hence the roots of the characteristic equation are generally complex as well.

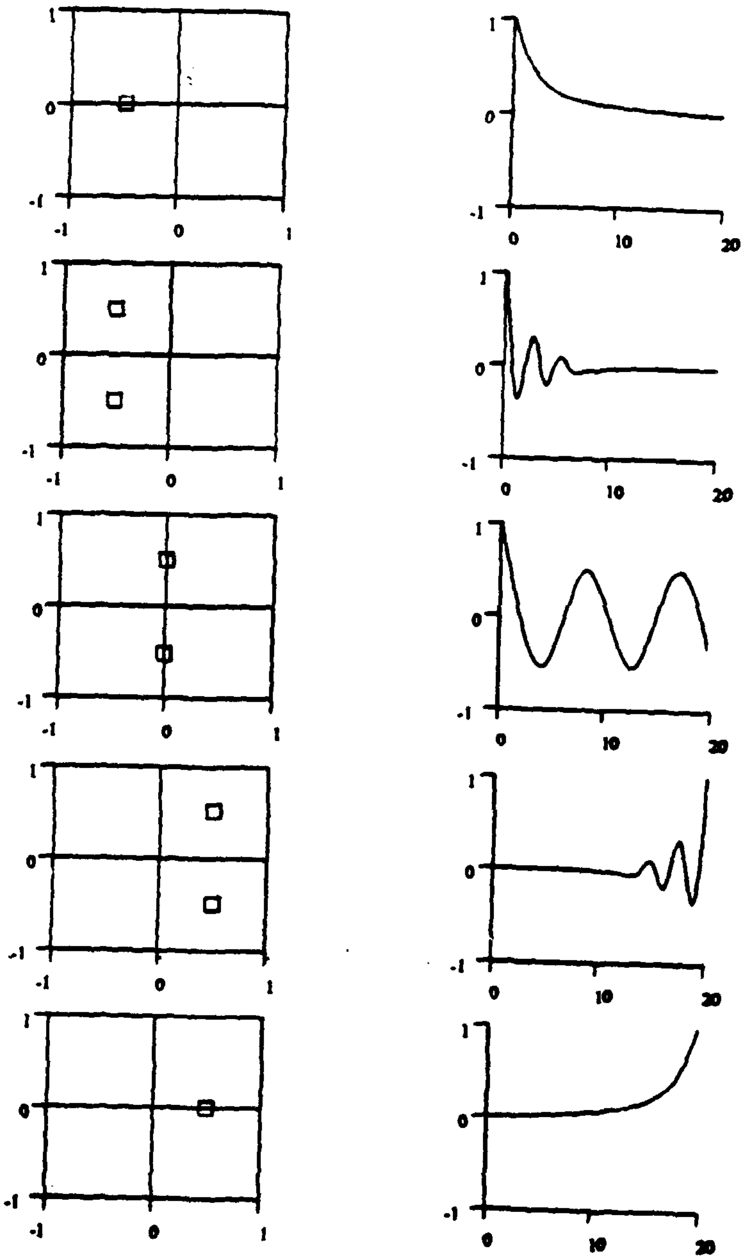


Figure 4.7: The connection between pole placement in the complex s-plane and performance in the time domain.

Figure 4.7 shows the relation between the poles of the transfer function for a second order system and the system's corresponding behavior in the time domain. In Figure 4.7, each plot on the left hand side indicates one particular placement of the poles in the complex s-plane, and the corresponding plot on the right indicates the resulting performance in the time domain. This method of analyzing control systems by determining the placement of poles is known as the *root locus* method.

Not surprisingly, there is close connection between the frequency- and time-domain methods for determining stability. In the case of multiple-input, multiple-output systems, we have to generalize on the notion of a transfer function, which is defined only for single-input, single-output systems. The *transfer matrix* of a linear multivariable dynamical system as introduced in the beginning of this section is uniquely defined by

$$T(s) = C(sI - A)^{-1} B,$$

where I is the identity matrix [29]. It should be noted that there is information lost in this conversion. In particular, the state and input equations specify the internal state as well as the input/output behavior of the dynamical system, whereas the transfer matrix only specifies the latter. It turns out that the poles of the system represented by the transfer matrix are exactly the eigenvalues of the matrix A [29].

One convenient property of transfer functions and transfer matrices is that, in certain cases, such representations can be obtained experimentally by subjecting the dynamical system to sinusoidal inputs and measuring the steady-state response. The close connection between frequency- and time-domain methods allows the engineer to shift back and forth between these two perspectives as the problem dictates.

Stability can simplify the design of control systems; it is not, however, a prerequisite for control. The linear system for the inverted pendulum is not stable, but it is controllable. If we are designing a device, it is generally worthwhile to design it in such a way that its corresponding dynamical system is stable. In cases in which the plant (environment) is given, we have little choice and must proceed whether or not the associated system is stable.

4.5 Optimality

In previous sections, we have stressed primarily the qualitative properties of dynamical systems (*e.g.*, controllability, observability, and stability). With the exception of criteria concerning whether or not a given controller can achieve a particular state from some arbitrary initial state, we have had very little to say about the performance of a control system. In this section, we consider control problems in which some quantitative measure (or *index*) of performance is provided. It is natural within this context to consider problems of *optimal control* that involve maximizing or minimizing such a performance index.

In describing optimal control problems, we generally restrict our attention to some restricted interval of time, either continuous, $[t_0, t_1]$, or discrete, $[1, n]$. The behavior of the dynamical system is described by either a set of differential equations

$$\dot{\mathbf{x}}(t) = f(x(t), u(t)), \text{ restricted to } t_0 \leq t \leq t_1$$

in the continuous case, or a set of difference equations

$$x(k + 1) = f(x(k), u(k)), \text{ restricted to } 1 \leq k \leq n$$

in the discrete case. In addition to the model for the dynamical system, it is often convenient to place restrictions on both the inputs (*e.g.*, you might want to place a bound on control torques to keep the cost of servo motors within budget constraints) and the outputs (*e.g.*, you may want to restrict the trajectories of a robot arm to a confined work space). The input restrictions define a set of *admissible controls* (see the discussion in Section 4.2 on admissible controls for discrete events systems). Finally, it will be necessary to formulate a performance index in terms of a scalar *value* function, V.

The choice of performance index is largely subjective, but generally a particular application will suggest something reasonable. In some cases, it may make sense simply to minimize time:

$$\mathrm{V} = \int_{t_1}^{t_2} 1\, dt = t_1 - t_0.$$

In other cases, there may be an obvious cost function, $c(x, u)$, such as the amount of fuel or other resource spent:

$$\mathrm{V} = \int_{t_1}^{t_2} c(x(t), u(t))\, dt.$$

For the set-point regulation and servo problems a good measure of performance is the squared error:

$$V = \int_{t_1}^{t_2} (x(t) - x^*(t))^2 \, dt.$$

where $x^*(t)$ is the desired state at time t. The squared error index is an example of a quadratic performance index.[4] More generally, the performance index is defined as

$$V = h(x(t_1)) + \int_{t_1}^{t_2} g(x(t), u(t)) \, dt.$$

where h and g are scalar functions meant to capture the value of the terminal state and the state/input trajectory respectively. The problem of designing optimal controls consists of finding an admissible control that minimizes (maximizes) the performance index, V.

There are two classes of optimal control problems involving linear multivariable systems for which general results have been obtained. The first class involves the use of a quadratic performance index as in the example of the minimum squared error index, and includes optimal versions of the linear set-point regulation and servo problems. In the second class of problems, the objective is to minimize the time required to drive the system to a desired state. In both of these two classes of problems, optimal controllers can make use of feedback, which, as covered in the next section, provides for more robust control in the presence of external disturbances and errors in modeling. The optimal linear minimum-time controller is of a particularly simple form: it can be viewed as a function that simply switches between the extreme values dictated by the class of admissable controls. A controller that operates at a constant level either in one mode or another (*e.g.*, $\forall t, \; u(t) \in \{-1, 0, 1\}$) is called a *bang-bang* controller.

Most of the work on optimal control builds upon basic techniques in the calculus of variations [12]. The method of Lagrange multipliers[5] for finding extrema of functions subject to constraints is one techniques from

[4] The function $V = \int f(t)\,dt$ is a quadratic performance index if $f(t) = x(t)'Ax(t)$ where A is an $n \times n$ matrix with $a_{ij} \in \mathbf{R}$ and $x \in \mathbf{R}^n$.

[5] Leonard Euler (1707–1783) developed the basic approach to solving constrained extremum problems. Joseph Lagrange (1736–1813) studied Euler's approach and worked out the details for some important special cases. The basic method is generally referred to as the method of *Lagrange multipliers*, but in some texts the equations are referred to as the *Euler-Lagrange equations* recognizing Euler's contributions.

the calculus of variations that students typically encounter in college calculus courses.

As a simple example illustrating the use of the method of Lagrange multipliers, let $\varphi(x, y)$ and $\varsigma(x, y)$ be functions of two variables. The object is to find values of x and y that maximize (or minimize) the *objective* function $\varphi(x, y)$ while at the same time satisfying the constraint equation, $\varsigma(x, y) = 0$. We replace $\varphi(x, y)$ with an auxiliary function of three variables called the *Hamiltonian* function, $\Phi(x, y, \lambda)$, defined as

$$\Phi(x, y, \lambda) = \varphi(x, y) + \lambda\varsigma(x, y).$$

The new variable, λ, is called a *Lagrange multiplier*. The *Euler-Lagrange multiplier theorem* [12] implies that, if we locate all points (x, y, λ) where the partial derivatives of $\Phi(x, y, \lambda)$ are all 0, then among the corresponding (x, y) we will find all of the points at which the function $\varphi(x, y)$ will have a constrained extremum.

In the method of *Lagrange multipliers*, we solve for x, y, and λ in the equations formed by setting the partial derivatives to 0:

$$\frac{\partial\Phi}{\partial x} = 0, \quad \frac{\partial\Phi}{\partial y} = 0, \text{ and } \frac{\partial\Phi}{\partial\lambda} = 0.$$

Since $\partial\Phi/\partial\lambda = \varsigma(x, y)$, if we find a solution (x, y, λ) to the above three equations, the constraint equation $\varsigma(x, y) = 0$ will automatically be satisfied.

To illustrate how to apply the method of Lagrange multipliers to problems in optimal control, consider the discrete-time system

$$x_{k+1} = f(x_k, u_k),$$

and the performance index defined by

$$V = \sum_{k=1}^{n} g(x_k, u_k),$$

where we have changed our notation somewhat, $x(k) = x_k$ and $u(k) = u_k$, to simplify subsequent equations. The only constraint that we impose is that the optimal solution obey the state difference equations. We enforce this constraint by augmenting the performance index as follows

$$V' = \sum_{k=1}^{n} \left[g(x_k, u_k) + \lambda_{k+1}(f(x_k, u_k) - x_{k+1})\right].$$

We define the Hamiltonian somewhat differently from above as

$$\Phi_k = g(x_k, u_k) + \lambda_{k+1} f(x_k, u_k),$$

so that we can rewrite the augmented performance index as

$$V' = \sum_{k=1}^{n} [\Phi_k - \lambda_{k+1} x_{k+1}].$$

By the Euler-Lagrange multiplier theorem, the change in the total derivative, dV', defined as

$$dV' = \sum_{k=1}^{n} \left[\left(\frac{\partial \Phi_k}{\partial x(k+1)} - \lambda_k \right) d\underline{x}_k + \left(\frac{\partial \Phi_k}{\partial \lambda_{k+1}} - x_k \right) d\underline{\lambda}_k + \frac{\partial \Phi_k}{\partial u(k)} d\underline{u}_k \right].$$

snould be zero at a constrained minimum. As a consequence, the necessary conditions for a constrained minimum are defined by

$$x_{k+1} = \frac{\partial \Phi_k}{\partial \lambda_{k+1}} = f(x_k, u_k), \quad 1 \le k \le n,$$

referred to as the *state equations*,

$$\lambda_k = \frac{\partial \Phi_k}{\partial x(k+1)}, \quad 1 \le k \le n,$$

referred to as the *costate equations*,

$$0 = \frac{\partial \Phi_k}{\partial u(k)}, \quad 1 \le k \le n$$

referred to as the *stationary conditions*, and, finally, we require that the x_1 be the initial state. The state and costate equations are coupled difference equations, and together they define a two-point boundary value problem. In the special case of linear systems with quadratic performance indices, numerical solutions can be obtained rather easily.[6]

In general, it can be quite difficult to solve the two-point boundary value problems resulting from Lagrange multiplier formulations. However, in some

[6] Specifically, it is possible to derive open-loop (the system state is not employed in computing the next input) controllers for the case in which the final state is specified (fixed) in advance, and closed-loop (the system state is employed in computing the next input) controllers for the case in which the final state is not specified (free) in advance [21].

cases, finding global maxima or minima can still be achieved by searching the space defined by the variational variables (*e.g.*, x and y in the case of minimizing $\varphi(x,y)$) One approach is to use numerical methods to solve the original equations relating to the performance index and constraints, and then search the resulting surface looking for global extrema. The *gradient*, defined as

$$\nabla\varphi = \begin{bmatrix} \partial\varphi/\partial x \\ \partial\varphi/\partial y \end{bmatrix}$$

in the case of $\varphi(x,y)$, is used to guide search in a method that proceeds by taking many small steps, each one in the direction indicated by the (negated) gradient. This search method is called *gradient descent*. If the surface has a single (global) minimum, then gradient descent search is guaranteed to find it. If, however, there are many local minima, as is often the case, then one has to be a lot more clever in directing the search. It is this aspect of optimal control involving search in a space of possible controls that primarily interests us in this section.

In some cases, we can resort to exhaustive search. For instance, if x and y are bounded, we might try to discretize the domain of φ, allowing each of x and y to take on $r \in \mathbf{Z}$ possible values. In this case, there are only r^2 points at which to evaluate φ; however, in the case of m variational variables each having r possible values, there will be r^m points to evaluate. As we will see, the dimensionality, m, of a control problem is a critical factor in the design of optimal control systems.

Bellman [3] and Pontryagin [24] were largely responsible for formulating the necessary problems and developing many of the basic approaches to solving optimal control problems. The requisite mathematics is complicated enough that the background required to even state the basic theorems does not seem warranted for our treatment here. Suffice it to say that the results for linear systems are extensive, and that, additionally, there are powerful numerical methods that have proved successful for a range of nonlinear systems. For a good overview of the field the reader is encouraged to consult the text by Athans and Falb [2]. In the remainder of this section, we focus on a particular class of optimal control problems called *multistage decision processes*, and a particular approach to solving such problems optimally *dynamic programming* due to Richard Bellman.

Consider a deterministic discrete-time n-stage process consisting of an initial state x_1, a sequence of inputs $u_1, u_2, \ldots, u_n$, and a sequence of resulting states $x_2, x_3, \ldots, x_n$ such that

$$x_{k+1} = f(x_k, u_k).$$

Following standard practice, the $\{u_k\}$ and $\{x_k\}$ are treated as variables ranging over U and X respectively. We introduce a performance index,

$$V(u_1, \ldots, u_n; x_1, \ldots, x_n).$$

We wish to find input sequences that maximize V.

As we indicated earlier, in general, this problem of maximizing a function of n variables is computationally quite hard. In the worst case, it will be necessary to search through the set of $|U|^n$ possible sequences of length n in order to choose the sequence with the highest value. In some cases, however, we can do much better. In the following, we consider some easier problems that result from introducing restrictions on V. In particular, we consider the case in which at a stage in the process, say the kth stage, the effect of the remaining $n-k$ stages on the total value depends only on the state of the system following the kth decision and the subsequent $n-k$ decisions [4]. Let $\mathrm{R} : U \times X \rightarrow \mathbf{R}$ represent a *reward* function, where $\mathrm{R}(u, x)$ corresponds to the (immediate) benefit derived from performing action u in state x. We write $\mathrm{R}(u, x)$ if both the input and the state matter in determining the amount of reward and $\mathrm{R}(x)$ if only the state matters. As an example of the sort of performance functions we are interested in, we might have

$$V(u_1, \ldots, u_n; x_1, \ldots, x_n) = \sum_{k=1}^{n} \mathrm{R}(u_k, x_k)$$

in which we are interested in the sum of rewards (referred to in the sequel as *separable control*), or

$$V(u_1, \ldots, u_n; x_1, \ldots, x_n) = \mathrm{R}(x_n)$$

in which we are interested only in the reward associated with the final state (referred to as *terminal control*).

We proceed by generating a sequence of functions, $\{V_n\}$, so that

$$V_n(x_1) = \max_{u_k} \sum_{k=1}^{n} \mathrm{R}(u_k, x_k).$$

Expanding, we have

$$\begin{aligned}
V_n(x_1) &= \max_{u_k} \sum_{k=1}^{n} R(u_k, x_k) \\
&= \max_{u_k}[R(u_1, x_1) + R(u_2, x_2) + \cdots + R(u_n, x_n)] \\
&= \max_{u_1} \max_{u_2} \ldots \max_{u_n}[R(u_1, x_1) + R(u_2, x_2) + \cdots + R(u_n, x_n)].
\end{aligned}$$

Rearranging, we obtain

$$\begin{aligned}
V_n(x_1) &= \max_{u_1}[R(u_1, x_1) + \\
&\quad \max_{u_2} \max_{u_3} \ldots \max_{u_n}[R(u_2, x_2) + R(u_3, x_3) + \cdots + R(u_n, x_n)]].
\end{aligned}$$

Note that

$$V_{n-1} = \max_{u_2} \max_{u_3} \ldots \max_{u_n}[R(u_2, x_2) + R(u_3, x_3) + \cdots + R(u_n, x_n)].$$

Substituting, we have in the case of separable control,

$$V_n(x_1) = \max_{u_1}[R(u_1, x_1) + V_{n-1}(x_2)],$$

or just

$$V_n(x) = \max_{u}[R(u, x) + V_{n-1}(f(x, u))]$$

for $n \geq 2$, and

$$V_1(x) = \max_{u} R(u, x).$$

for $n = 1$. For the case of terminal control, we have

$$V_n(x) = \max_{u}[V_{n-1}(f(x, u))], \quad \text{for} \quad n = 2, 3, \ldots$$

and

$$V_1(x) = R(x).$$

The time to compute $V_i(x)$ for all $x \in X$ given that invoking V_{i-1} has unit cost is $O(|X||U|)$. From this observation, it follows that the time required to compute $V_n(x)$ for all $x \in X$ given that invoking V_1 has unit cost is $O(n|X||U|)$.

This general method of computing the performance index recursively is called *dynamic programming.* The basic constrained minimization variational problem essentially involves choosing a point in an n-dimensional

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X															X
X															X
X				X											X
X				X											X
X		G		X											X
X				X											X
X				X				X	X	X	X				X
X								X							X
X								X							X
X								X							X
X								X	X	X	X				X
X															X
X															X
X															X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 4.8: A 16 × 16 grid world

phase space. Dynamic programming involves decomposing the problem into making n choices each of which involves a one-dimensional phase space [4].

To illustrate the basic technique involved in dynamic programming, we consider a simple robot control problem. A grid world is represented as an $n \times n$ grid. One cell of the grid is designated as the goal. Certain other cells (a total of m) are designated as obstacles. In particular, all of the perimeter cells are designated as obstacles. Initially, the robot is located in a cell which is not an obstacle. Figure 4.8 depicts a 16 × 16 grid world in which the goal is indicated by G and the obstacles by X's. fonts

There are $n^2 - m$ states each one corresponding to the robot being in a particular cell not designated as an obstacle. There are ~~nine~~ possible actions not all of which are necessarily available for a given state: the robot can remain in its current cell or move to any one of four adjacent cells (|. —. |, and ←) as long as the destination cell is not designated as an obstacle. We use the value function for separable control where the reward is defined as five ✓

$$R(u, x) = \begin{cases} 0 & \text{if } x \text{ is equal to the goal} \\ -1 & \text{otherwise} \end{cases}$$

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X	-5	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	X
X	-4	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	X
X	-3	-2	-3	X	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	X
X	-2	-1	-2	X	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	X
X	-1	G	-1	X	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	X
X	-2	-1	-2	X	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	X
X	-3	-2	-3	X	-7	-8	-9	X	X	X	X	-16	-17	-18	X
X	-4	-3	-4	-5	-6	-7	-8	X	-20	-19	-18	-17	-18	-19	X
X	-5	-4	-5	-6	-7	-8	-9	X	-21	-20	-19	-18	-19	-20	X
X	-6	-5	-6	-7	-8	-9	-10	X	-22	-21	-20	-19	-20	-21	X
X	-7	-6	-7	-8	-9	-10	-11	X	X	X	X	-18	-19	-20	X
X	-8	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	X
X	-9	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	X
X	-10	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 4.9: V($\langle x, y\rangle$) for the Grid world

We compute V_1, V_2, up to V_k such that $V_i = V_{i-1}$ and set $V = V_i$. Figure 4.9 shows V($\langle x, y\rangle$) for each state (location $\langle x, y\rangle$) in the grid world of Figure 4.8.

If you look carefully at the numbers shown in Figure 4.9, you will notice that by always moving to the neighboring location with the highest value you will eventually end up at the goal location no matter what location you start out in. This property can illustrated graphically by considering the elevation map shown in Figure 4.10 defined using V($\langle x, y\rangle$) as the elevation at coordinates $\langle x, y\rangle$ in the grid with interior obstacles represented as small negative values. Notice that the goal location is a global maximum in the elevation map. This will always be the case no matter what the arrangement of obstacles. It turns out that the strategy of always moving to the location with the highest value is optimal in the following sense.

We define a control law or *policy* as a mapping from states to actions:

$$\eta : X \rightarrow U.$$

We are interested in policies that are optimal according to the following principle of Bellman. "Principle of optimality. An optimal policy has the

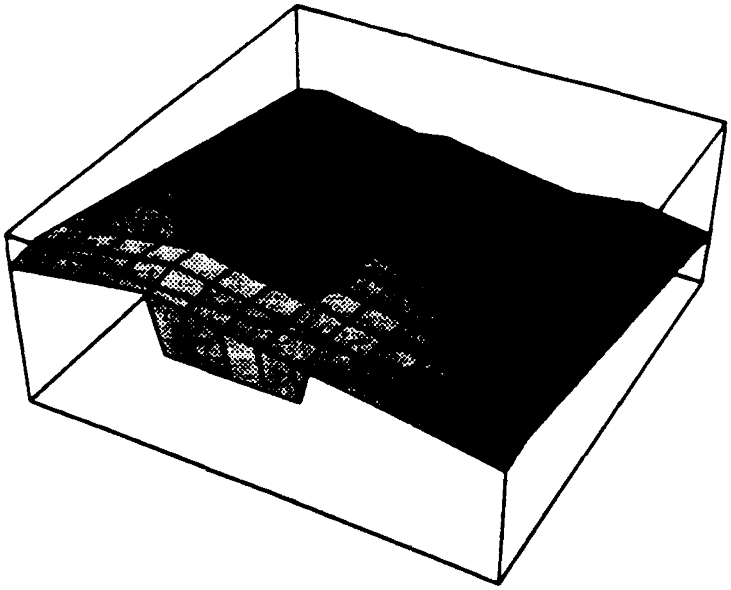


Figure 4.10: Representation of $V(\langle x,y\rangle)$ as an elevation map

property that whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." ([4] pg. 57) Given Bellman's principle of optimality, the following policy

$$\eta(x) = \arg\max_{u} V(f(x,u))$$

is optimal.

Figure 4.11 shows the optimal policy for the grid world shown in Figure 4.8, where —. —, ↑, and ↓ indicate the direction of movement for the indicated state as specified by the optimal policy.

Because the transitions in state space are so localized in the grid world, we can use a much more efficient dynamic programming algorithm for computing the optimal policy than the one described above. In particular, we compute V_2 only for grid cells corresponding to one of the four neighbors of the goal adjacent along the grid axes, and, in so doing, treat V_1 as undefined for all cells other than the goal. In general, we compute V_i only for previously unconsidered grid cells corresponding to one of the four neighbors of cells considered in $i-1$th iteration, and treat V_{i-1} as undefined for all

X	X	X	:	X	X	X	X	X	X	X	X	X	X	X	X
X	↓	↓	←	←	←	←	←	←	←	←	←	←	←	←	X
X	↓	↓	←	←	←	←	←	←	←	←	←	←	←	←	X
X	↓	↓	←	X	↑	←	←	←	←	←	←	←	←	←	X
X	↓	↓	←	X	↑	←	←	←	←	←	←	←	←	←	X
X	→	G	←	X	↑	←	←	←	←	←	←	←	←	←	X
X	↑	↑	←	X	↑	←	←	←	←	←	←	←	←	←	X
X	↑	↑	←	X	↓	←	←	X	X	X	X	↑	←	←	X
X	↑	↑	←	←	←	←	←	X	→	→	→	↑	←	←	X
X	↑	↑	←	←	←	←	←	X	↑	↑	↑	↑	←	←	X
X	↑	↑	←	←	←	←	←	X	↑	↑	↑	↑	←	←	X
X	↑	↑	←	←	←	←	←	X	X	X	X	↓	←	←	X
X	↑	↑	←	←	←	←	←	←	←	←	←	←	←	←	X
X	↑	↑	←	←	←	←	←	←	←	←	←	←	←	←	X
X	↑	↑	←	←	←	←	←	←	←	←	←	←	←	←	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 4.11: An optimal policy

cells not considered in the $i-1$ or earlier iterations. If k is the last iteration in which there are unconsidered cells, then V_k is defined for all cells in the grid, and we set $V = V_k$. This specialized dynamic programming algorithm runs in $O(|X|)$.

The example application of dynamic programming given above involves a discrete deterministic dynamical system. Dynamic programming can be applied to continuous dynamical systems to achieve solutions of arbitrary accuracy using a variety of numerical techniques. Dynamic programming can be seen as a method of efficiently solving variational problems involving multiple local minima by cleverly guiding the search. Dynamic programming can also be applied to stochastic processes, and we will return to this subject in Chapter 6.

Here as elsewhere the dimensionality of the problem severely restricts the application of this and most other methods to generating solutions efficiently. Dynamic programming is often referred to as an "approach" rather than a "method," where the distinction generally made is that an approach provides a way of looking at problems that still requires considerable creativity to actually apply, whereas a method is more a matter of turning a crank. Dynamic programming suggests that we try to view optimization problems as multistage decision problems in which the performance index is some simple (*e.g.*. additive) function of the state and input at each stage. If it is possible to view a problem thus. we can effectively reduce the dimensionality of the problem thereby. availing ourselves of substantial computational savings. Unfortunately, there are many aspects of a problem that serve to determine its dimensionality. For example, at best, the solution methods that we considered above involved computations linear in the size of the state space. and the dimensionality of the state space is determined by the number of state variables that comprise the state vector. In practical problems. methods that require quantifying over the entire state space can be computationally prohibitive. In subsequent chapters. we consider methods that will allow us to decompose certain problems into independent subproblems each of which requires quantifying over only a small portion of the state space.

4.6 Feedback Control Systems

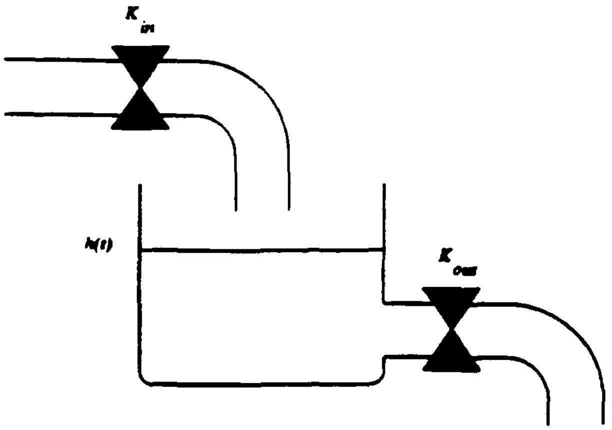
In Section 4.2 on controllability, we considered a controller as a function from states to inputs (control actions). While there are many different types of controllers mentioned in the literature, this particular formulation is perhaps the most common. It is so common. in fact, that traditionally a *control law* is defined to be a function $\eta : T \times X \to U$,

$$u(t) = \eta(x(t), t).$$

However. in the problems we will be considering, η will not depend on the current time.

This basic idea that the inputs to a dynamical system should be computed from the state is quite important. Kalman describes it as "the fundamental idea of control theory," and "a scientific explanation of the great invention known as 'feedback,' which is the foundation of control engineering" ([16] pg. 46).

It is worth asking why, if we have an accurate model of the process that we are trying to control. must we resort to sampling the state of this process on a continual basis. The answer is that uncertainty can and. generally. does arise from several sources besides the dynamical model. For instance. we have to sample the state of the system at some point in order to supply the initial conditions to the model. If there is any error in our measurement



???

Figure 4.12: Controlling the level of fluid in a tank

of the state variables, then that error will likely be exacerbated with the passage of time and as a consequence of inappropriate inputs generated on the basis of incorrect state information. Even if we are able to observe the state precisely, there will inevitably be some delay between our observation of the state and our initiation of a control action. This delay may be due to time spent in computing inputs, the response time of the actuators used to realize an input, or lags introduced by the sensors. We return to these issues in Chapter 6 when we consider the problems that arise in dealing with uncertainty in control.

In the following, we consider the application of feedback control to some of the problems introduced in Chapter 1. We begin by considering the problem of regulating the level of fluid in a tank using a closed-loop feedback controller. Figure 4.12 depicts the tank and its associated input and output pipes.

2

We model the controlled process as a first-order differential equation:

$$K_{in}\theta(t) - K_{out}h(t) = A\frac{dh(t)}{dt}$$

where K_{in} is the flow constant in cubic meters per degree minute for the valve governing flow through the input pipe, K_{out} is the flow constant in square meters per minute for the output pipe. A is the surface area of the

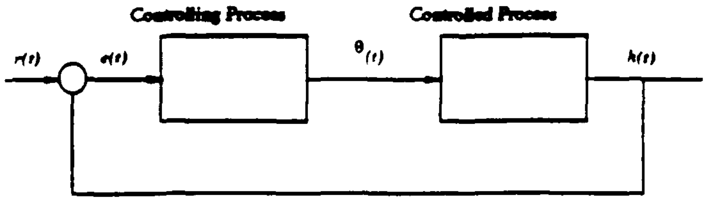


Figure 4.13: Block diagram for a closed-loop process controller

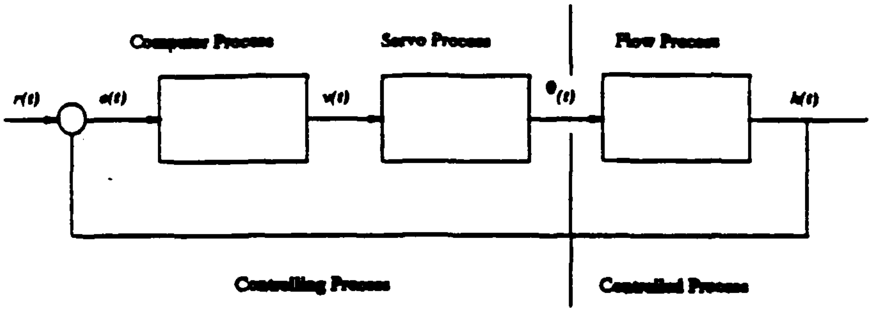


Figure 4.14: Decomposing the controlling process into subprocesses

tank. $\theta(t)$ is the position of the valve governing flow through the input pipe at time t, and $h(t)$ is the height of the fluid in the tank at time t.

Now we have to specify a controlling process that changes θ in order to cause changes in h. In the simplest model, the controlling process directly determines θ by looking at the difference between the reference (or target) level and last measured value of h; this difference is referred to as the *error*. The block diagram shown in Figure 4.13 depicts this model with $r(t)$ indicating the reference and $e(t)$ indicating the error.

In Chapter 1, we defined a control algorithm that could cause instantaneous changes in θ. Needless to say, the typical interface between the controlling and controlled processes is more complex. In a somewhat more realistic model, the control computer might determine a voltage that is input to a servo system consisting of an amplifier and a DC motor attached to the input valve. The servo system is just another process, and we might model it using the equation:

$$\frac{d\theta(t)}{dt} = K_g\, v(t)$$

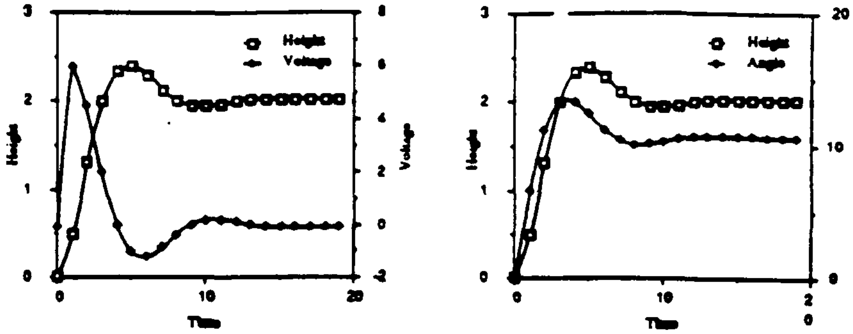


Figure 4.15: The behavior of the discrete proportional controller

where $v(t)$ is the input voltage and K_g is a constant that depends on the characteristics of the servo. Figure 4.14 provides a block diagram of this more complex model.

To define a process that determines the voltage input to the servo, we employ a standard technique from control theory. In many control schemes, the output of the controller is a simple function of the error. For controlling certain processes, an effective controller can be designed in which the output of the controller, $v(t)$ in this case, is directly proportional to the error:

$$v(t) = K_p e(t)$$

where K_p represents the controller proportionality constant. Not surprisingly, this sort of control is called *proportional control*.

For a control algorithm running on a digital computer, we have to specify a *discrete controller* that samples the output of the controlled process and outputs a control action at discrete intervals. The discrete proportional controller is just a computer program running on a specific machine that samples the output of the controlled process every so many clock cycles and outputs a value proportional to the computed error.

To maintain the level of fluid in the tank depicted in Figure 4.12 at two meters, we might use the following loop:

```
while true
    wait_for_delay;
    height — read_fluid_height;
    error — 2.0 - height;
    servo_voltage — $K_p$ * error;
```

where **read_fluid_height** reads the height sensor, **wait_for_delay** causes the controller to pause for the specified sample period, and **servo_voltage** is a machine register that directly determines the voltage fed to the servo. Figure 4.15 shows two graphs describing the behavior of the above control algorithm with a sample period of 1 minute and a proportionality constant of 3.0. One graph compares changes in h with changes in r, and a second compares changes in h with changes in θ. The particular proportionality constant 3.0 was chosen after a small amount of experimentation.

Proportional controllers are suitable for controlling only a limited class of processes. Two other popular forms of control are *integral control* and *derivative control*. The output $u(t)$ of an integral controller is proportional to the accumulated error:

$$u(t) = K_i \int_0^t e(t)\,dt$$

whereas the output of a derivative controller is proportional to the change in the error:

$$u(t) = K_d \frac{d\,e(t)}{dt}.$$

The proportional-plus-integral-plus-derivative (or *PID*) controller generalizes the above three types of controllers:

$$u(t) = K_p\,e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{d\,e(t)}{dt}$$

For the simple tank-filling process, proportional control is quite adequate. Other, less stable processes, such as the inverted pendulum introduced in Chapter 1, may require an integrator and a differentiator to damp oscillations and compensate for abrupt disturbances.

It should be noted that the constants used in a discrete *PID* controller are dependent upon the sample period. Of course, once you have the coefficients for the continuous *PID* controller you can derive the coefficients for a discrete controller of any sample period.

The mathematical discipline of control theory is largely concerned with the formal analysis of control systems. As was mentioned in Section 4.5, in some cases, optimal control processes can be derived analytically providing that accurate models of the controlled processes are available. Since the characteristics of the controlled processes rarely are known precisely, control theorists are interested in systems that are insensitive to minor deviations in the models used in the design process. In cases where significant deviations are likely, or the models are known to be incomplete, *adaptive systems* are designed to compensate by adjusting the model as information becomes available.

Adaptive control techniques attempt to cope with uncertainty about the process being controlled by automating certain aspects of controller design. The basic idea is quite simple. The designer generally has some sort of model of the process or *plant* that he is trying to build a controller for. This model, while it is known to provide only a rough idea of the behavior of the plant, is sufficient to determine the form of the basic controller (*e.g.*, a parameterized *PID* controller). The designer then builds a program that refines the basic controller as it observes this controller attempting to control the plant. In the case of a *PID* controller, refinement consists of adjusting the control coefficients. Adaptive control is one approach to making controllers more responsive to a complex and often unpredictable environment. Adaptive control also provides a means for coping with complexity in the design process by allowing a control system to monitor its own behavior and adjust accordingly. Chapter 9 deals with some aspects of adaptive control in the context of a discussion of learning techniques. Now we turn our attention to some more practical issues in building control systems.

¶ new section!

Control systems are complex devices that involve the interaction of mechanical and computational processes. In considering the computational aspects of control, it is important to keep in mind that someone has to write the programs or design the circuits that perform the necessary computations. For problems like controlling a power plant or an automated assembly line, these programs and circuits can become quite complex. Despite our best efforts, large programs develop organically as a process only partly under the control of any one individual. Continual redesign is impractical, and sooner or later the designer has to commit to a specific implementation of a module, interface, or subroutine. Once in a while, a designer has the luxury of rewriting an interface, optimizing an algorithm, or consolidating several functions in a single module, but often enough he or she has to make do with whatever is available. It would be convenient if control knowledge could be encapsulated in small general-purpose functional units that could be applied in a wide variety of circumstances. This has long been a dream of researchers in artificial intelligence, and, in the following, we consider some possible approaches to realizing that dream. Two critical issues that have to be addressed in the context of controlling processes are:

1. Can general-purpose control knowledge be used to support real-time control of interesting processes?

2. Can disparate behaviors be made to cooperate so as to achieve coordinated behavior across a range of situations?

In attempting to address these issues, we consider a class of programming techniques called *reactive systems* that were specifically designed to address shortcomings in classical approaches to planning relying primarily on off-line computation and perfect information. Reactive systems are meant to be responsive to the processes being controlled. They tend not to employ any complicated predictive mechanisms in order to avoid the computational overhead generally associated with such mechanisms. A reactive system has to be prepared to respond quickly to changes perceived in the controlled process. If the system is engaged in a complex and time-consuming computation, it will likely miss opportunities to generate appropriate responses. In the applications for which reactive systems are best suited, it should be possible to achieve the desired behavior using simple models that can be quickly computed.

Much of the work on reactive systems done in artificial intelligence has been concerned with building systems that are capable of representing and manipulating precompiled procedural knowledge about how to control things. Different behaviors can be separately realized in terms of distinct procedures each making use of the available sensors and effectors as needed. The differences between such systems usually revolve around the complexity of the primitive operations allowed by a given procedure and the means whereby procedures are selected, coordinated, and allowed to communicate with one another. In the following, we consider two approaches to building reactive systems. For the most part, the two approaches look like programming languages, and our analysis concerns what features of the different languages make them more or less suitable for writing and thinking about control systems.

Every programming language is designed to support a particular level of abstraction. High-level languages can introduce barriers to abstraction by forcing the programmer to adopt a particular way of thinking. For instance, a language that provides only sequential control constructs can make it difficult to deal with parallel or asynchronous processes. Low-level languages can also introduce barriers to abstraction simply by failing to provide the programmer with adequate means to deal with the complexity of programming large systems. Of course, one can simulate any computational process given any Turing-equivalent machine/language combination. In looking at approaches designed to facilitate controlling processes, we should be alert to notice features that allow us to naturally map our understanding of control problems onto computational processes.

Almost every programming language provides support for procedures of one sort or another. Procedures encapsulate procedural knowledge: how to go about achieving certain tasks. In speaking about the control of processes, procedures are usually associated with specific behaviors. The first approach to implementing reactive systems that we look at is called a *procedural reasoning system* [13]. A procedural reasoning system consists of a set of procedures and a *scheduler* for selecting what procedures to run and when. Each procedure has associated with it a specific task-achieving behavior that it implements, and an invocation condition or *goal* specifying what the procedure is meant to achieve.

Procedures are represented as *labeled transition graphs*. A labeled transition graph is a directed graph whose arcs are labeled with statements in some logic or programming language. In the following, we use Prolog statements to label arcs. The statements are examined by the scheduler to determine transitions from one node in the graph to some adjacent node in the graph. Each node in a labeled transition graph has one or more arcs leading out of it. Some statements correspond to predicates or queries and others have an imperative content. The statements labeling arcs are generally seen as giving rise to the goals of the system.

The scheduler is charged with keeping track of what goals the system has and invoking whatever procedures are appropriate to achieving those goals. At any given moment, the scheduler has some number of active procedures that it is employing to pursue its present goals. For each of those procedures, the scheduler maintains a pointer to some node in the associated labeled transition graph. The scheduler chooses a particular procedure to work on and attempts to transit to a new node by examining the statements on the arcs leading out of the node currently associated with the chosen procedure. An example should help clarify.

Figure 4.16 shows a labeled transition graph implementing the discrete proportional controller discussed earlier. The procedure shown also implements an overflow test to issue an alarm if the fluid runs over the top of the tank. Statements labeling arcs such as `fluid_height(Tank,Height).` and `V is K * (Target - Height)` correspond to queries: "what is the current height of the fluid in the tank?" and "what voltage is `K` times the difference between the current height and reference value?" Statements such as `set_servo_voltage(Tank,V)` and `set_alarm(Tank,1)` correspond to imperatives to adjust parameters used by the procedures associated with the

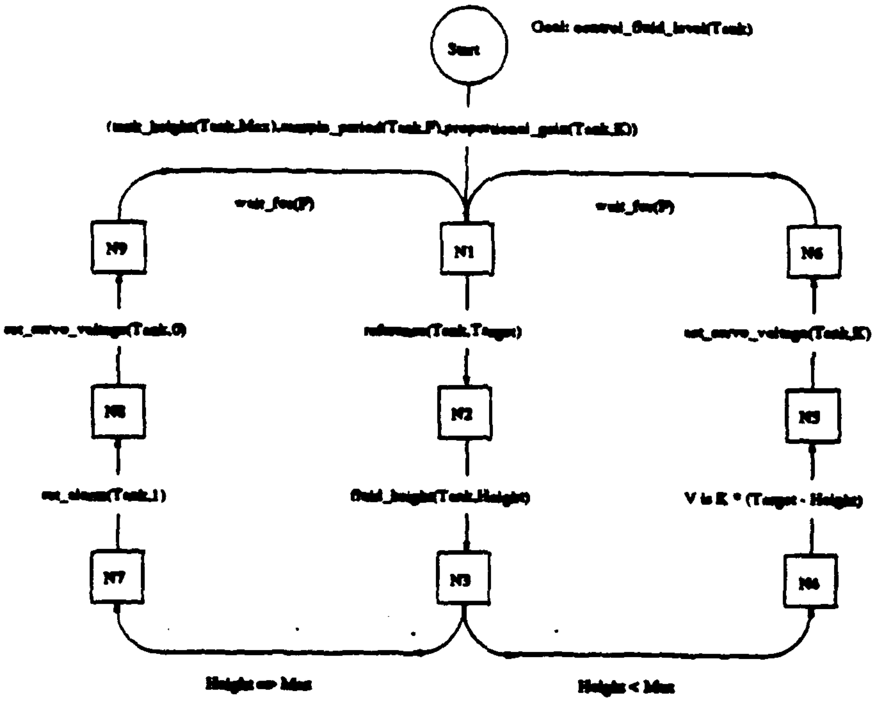

Start
Goal: control_fluid_level(Tank)
(tank_height(Tank,Max),sample_period(Tank,P),proportional_gain(Tank,K))
wait_for(P)
wait_for(P)
N9
N1
N6
set_servo_voltage(Tank,0)
reference(Tank,Target)
set_servo_voltage(Tank,K)
N8
N2
N5
set_alarm(Tank,1)
fluid_height(Tank,Height)
V is K * (Target - Height)
N7
N3
N4
Height => Max
Height < Max

Figure 4.16: Labeled transition graph for a proportional controller

\textwidth ✓

servo attached to the input valve and the alarm device.

Both queries and imperatives can be seen as giving rise to additional goals. For some of these goals, the scheduler invokes additional procedures. For other goals, special-purpose systems may kick in to try to satisfy the goal. For a given goal there may be many different procedures running. A procedure can be revoked if its associated goal becomes satisfied or if some competing goal becomes satisfied. Most labeled transition graphs have terminal nodes indicating exit conditions for the associated procedure. The scheduler is responsible for starting new procedures and terminating old ones. Procedures communicate with one another by posting goals to a global database in a manner similar to that used in blackboard systems [15]. A possible scheduling algorithm for a procedural reasoning system is ~~as~~ described as follows. The scheduler maintains two queues ACTIVE and PENDING to keep track of procedures that are in various stages of processing.

1. Choose a procedure p from ACTIVE.
2. Post goals corresponding to each statement labeling an arc emanating from the current node of the procedure p.
3. Move p from ACTIVE to PENDING.
4. Add to ACTIVE each procedure whose invocation condition matches a goal posted in Step 2.
5. For each procedure q in PENDING such that any of the posted goals corresponding to the statements labeling arcs emanating from the current node of q are satisfied:
   (a) Choose one satisfied goal g.
   (b) Retract the other posted goals and remove any associated procedures from ACTIVE and PENDING.
   (c) Set the current node of q to be the node terminating the arc labeled with the statement corresponding to g.
   (d) Remove q from PENDING.
   (e) If the current node of q is not a terminal node, move q to ACTIVE.
6. Go to Step 1.

It is important to note that the scheduler never waits around to compute anything: the scheduler simply posts new goals. invokes procedures where required. and notices when posted goals are satisfied. Suppose that the procedure shown in Figure 4.16 is the only active procedure and its current node is **N2**. The scheduler posts the goal **fluid_height(Tank,Height)** with **Tank** bound and **Height** unbound. and the procedure is moved to the list of pending procedures. The subsystem responsible for monitoring the level of fluid in the tank notices the posted goal. reads the sensor for fluid level. and marks the goal **fluid_height(Tank,Height)** as satisfied with **Height** bound to whatever the sensor read. The next time the scheduler looks at the pending procedures it notices the satisfied goal. updates the procedure's current node to **N3**. and places the procedure back on the list of active procedures.

The procedural reasoning system supports subroutine calls in that a transition in one procedure may require invoking a second procedure. Several procedures can run in parallel and communicate asynchronously by posting goals to the global database. As an example of how two procedures might work together in parallel. we consider a type of feedforward control that can be implemented easily in a procedural reasoning system.

The reference or target value specified in a control problem can be thought of as a command for the controller to achieve a particular condition (*e.g.*, a fluid level of the specified height). In many problems. the reference changes—sometimes continuously—over an interval. The controller has to track these changes so as to minimize errors. If the reference changes can be predicted or are simply provided in advance. the controller can take advantage of this to help eliminate certain errors by using feedforward control. reduce
For example. if the controller for a robot arm knows the exact trajectory it is to move the end effector along, it can often precompute a sequence of control actions. and then execute an error-free path without any feedback control whatsoever. In most cases. however, feedforward and feedback are used in conjunction, with feedforward taking advantage of known changes in the target value, and feedback compensating for the inevitable errors that arise
~~crop up~~ in dealing with real-world processes.

In the case of our tank-filling process. a feedforward controller could be added to the feedback controller of Figure 4.14. The feedforward controller anticipates the next reference value and mediates the output of the feedback controller if a change is detected. This sort of controller is referred to as a *command feedforward* controller and its block diagram is shown in Figure 4.17.

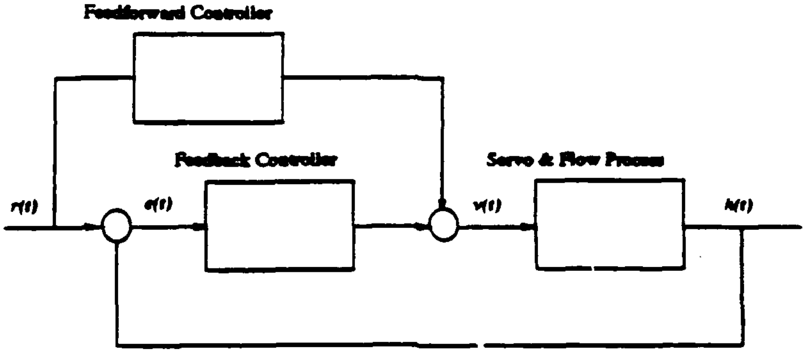


Figure 4.17: Block diagram for a controller with command feedforward

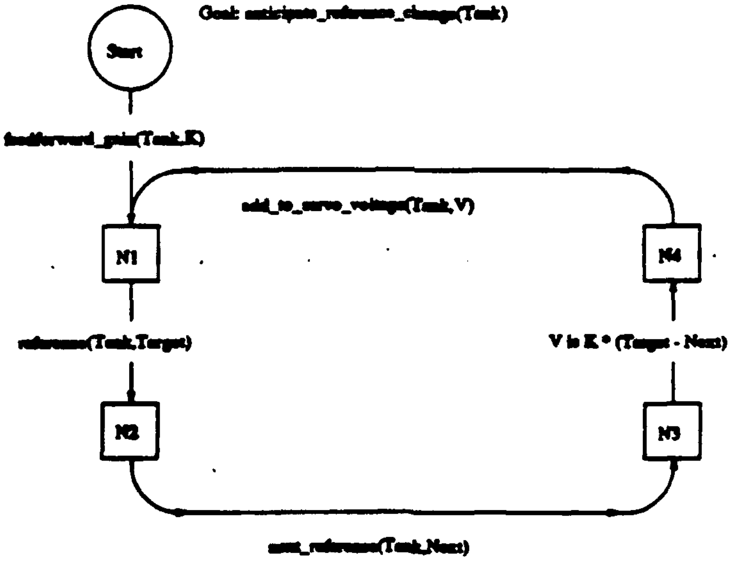
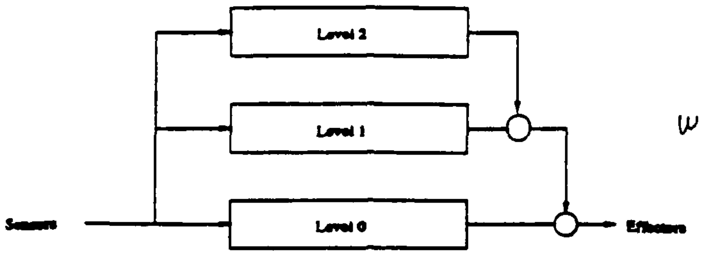

Goal: anticipate_reference_change(Tank)
Start
feedforward_gain(Tank,K)
add_to_servo_voltage(Tank,V)
N1
N4
reference(Tank,Target)
V is K * (Target - Next)
N2
N3
next_reference(Tank,Next)

Figure 4.18: Labeled transition graph for a command feedforward controller



wider

Figure 4.19: A hierarchical control system

To implement command feedforward control in a procedural reasoning system, we define a new procedure to monitor changes in the reference value. This procedure specifies a value proportional to the change in reference to be added to that specified by the feedback controller. The labeled transition graph for the command feedforward procedure is shown in Figure 4.18. The two procedures shown in Figure 4.16 and Figure 4.18 run at the same 'ime. The servo process operates on a voltage which is the sum of that specified by each of the two procedures. This control scheme works particularly well for tracking a continuously changing reference: for instance, if you wanted the level in the tank to decrease to 0 at a fixed rate.

In describing the command feedforward control system above, we started with an existing feedback control system and then added a feedforward controller, without changing the basic architecture of the feedback control system. ~~A~~ *Hierarchical control systems* generalize~~s~~ on this basic idea. A hierarchical control system is constructed of several layers so that each layer serves as a controller for the layer immediately below and is controlled by the layer immediately above. There are different types of hierarchical control systems. They differ in how the various layers are controlled by and impose control on the layers immediately above and below. As our second approach to building reactive systems, we consider a hierarchical control system in which one layer is allowed to impose control on a lower layer by modifying control signals used for communicating between components of the lower layer [7].

Figure 4.19 depicts the general form of the sort of hierarchical control system we are considering. Each level is composed of a set of components each of which is responsible for a simple primitive behavior. The components communicate with one another by passing signals. For the most part, the signals consist of bit or byte streams. The components can be implemented

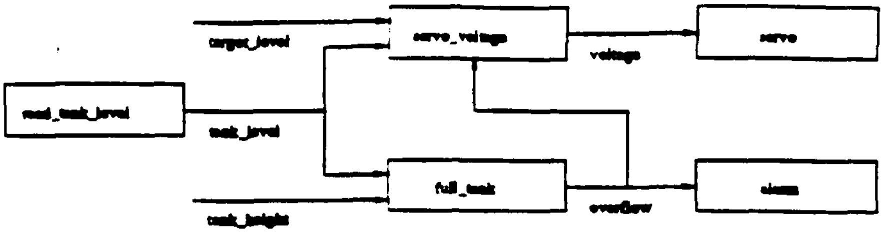


Figure 4.20: A single-level control system

any way that you want, but it is a good discipline to think of them as very simple computing devices. For instance, the components might be implemented as regular finite state machines augmented with a small amount of local state, a combinatorial circuit, and a local clock. The combinatorial circuit and local state are used to keep track of signals originating from other components. The clock is used to provide simple timing capabilities. There is no global state and the different components communicate asynchronously by writing values into the local memory of other components.

Figure 4.20 shows a single-level control system for ~~the~~ maintaining the fluid level in a holding tank. The component labeled **read_tank_level** continuously samples the sensor indicating the level of fluid in the holding tank and outputs the value read on the wire labeled **tank_level** which subsequently appears in registers in the components labeled **servo_voltage** and **full_tank**. The **servo_voltage** component implements the same procedure as the labeled transition graph of Figure 4.16. The **full_tank** component detects when the level in the tank is equal to the height of the tank and passes this information on to the the **servo_voltage** component and to the **alarm** component which is responsible for sounding an alarm.

To illustrate how one level in a hierarchical control system might influence a lower level in the same system, we consider a second form of feedforward control referred to as *disturbance feedforward* control. A disturbance is a process that affects the controlled process but is not taken into account by the controlled process model. In the fluid-level process we have been considering, we might model a process restricting the flow through the pipe leading out of the tank shown in Figure 4.12 as a disturbance. Suppose that the output pipe is being used to fill containers that are moved into position under the pipe using a conveyor system. When a container is filled, the flow through the output pipe is temporarily restricted so that a new container can be positioned under the pipe. Figure 4.21 shows how a simple propor-

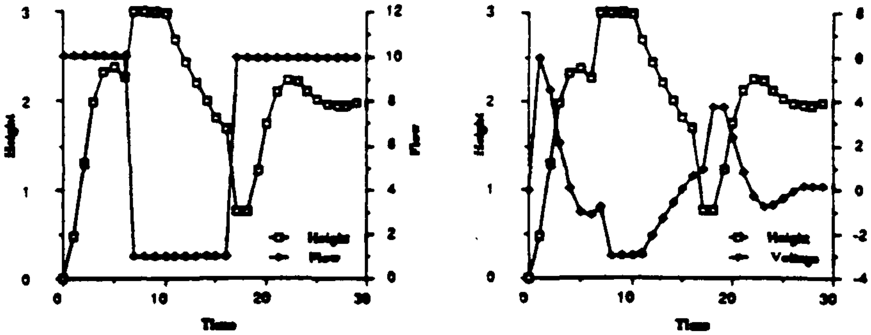
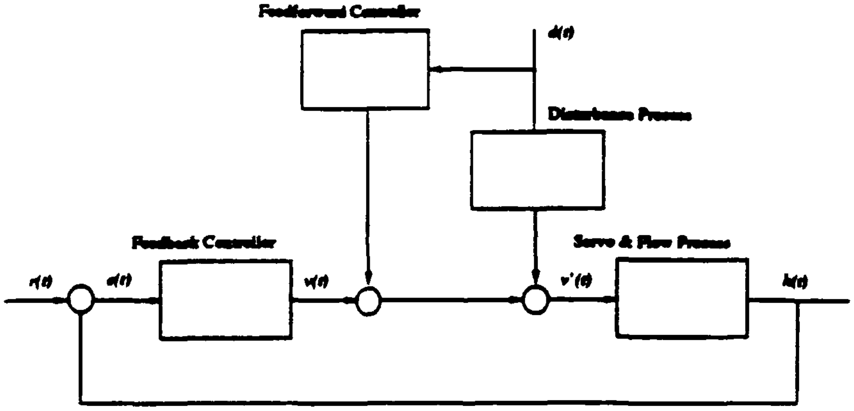


Figure 4.21: Overflow due to a disturbance restricting outflow



| widx |

Figure 4.22: Block diagram for a controller with disturbance feedforward

tional controller reacts to a brief restriction in the output flow: the reduced flow effectively reduces the gain of the proportional controller and fluid spills over the top of the tank before the controller can react and appropriately compensate.

Let us suppose that it is possible to anticipate a restriction in the output flow as would be the case for the container-filling example described above. Figure 4.22 shows a block diagram for a disturbance feedforward controller for the fluid-level problem. We assume that it is possible to sense restrictions in the output flow and use this information to increase the voltage fed to the servo motor thereby temporarily increasing the gain of the feedback controller.

Given the single-level proportional controller shown in Figure 4.20, we

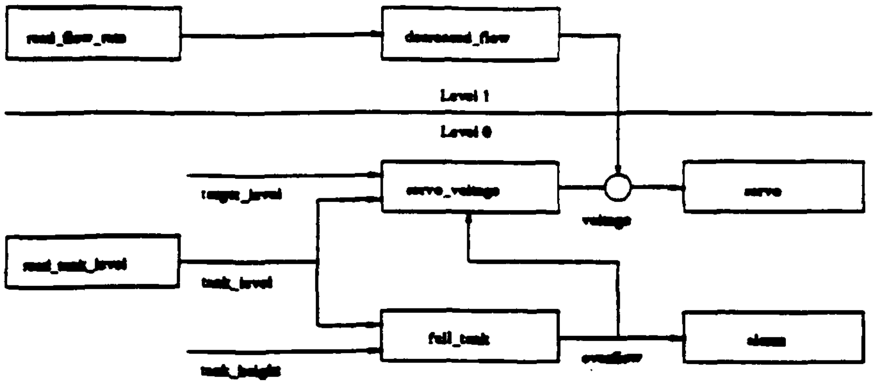


Figure 4.23: A two-level system with disturbance feedforward control

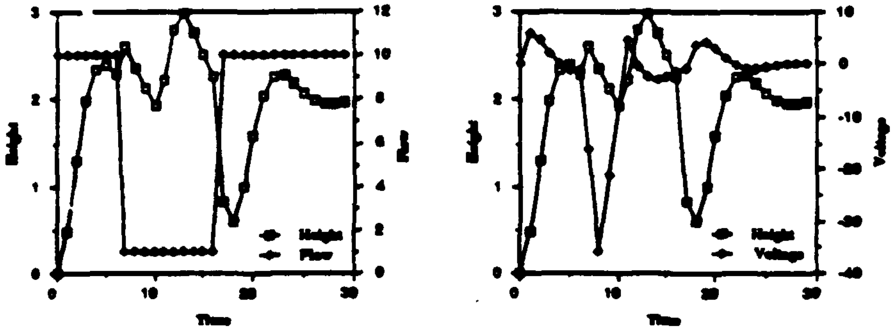


Figure 4.24: Disturbance feedforward controller preventing overflow

can add a second control level in order to reduce or eliminate the amount of spillage resulting from momentary restrictions. The resulting two-level system is shown in Figure 4.23.

The performance of the two-level system is somewhat less than optimal: as indicated in Figure 4.24, the two-level system does avoid spilling any fluid, but the fluid height is somewhat erratic around the time of the restriction. We might be able to further tune the feedforward component to eliminate or reduce this erratic behavior. However, it is often the case that, in building on top of an existing control system, we simply have to accept the limitations of what we started out with, or do it over. The hierarchical system described above makes it rather easy to build on an existing control system. Given the discipline described earlier for building modular stand-alone computational components, adding new functionality or enhancing old often consists of simply adding some new components and wiring them together with the old ones. To the extent that this can be realized in practice. it makes building and experimenting with control systems remarkably easy.

The procedural reasoning system and the hierarchical control system described above are similar in many respects. Both support multiple processes running in parallel. Both support procedural abstraction and asynchronous control. There are some differences. however. The procedural reasoning system encourages the explicit representation of intentions. behaviors. and goals. The hierarchical control system encourages one to think in terms of evolving control systems and distributed computation. We say "encourage" as both systems are no more than general-purpose programming languages. Unless you specify a compiler and a target machine. the two systems are essentially equivalent.

There are other approaches to building reactive systems some of which will be discussed in subsequent chapters. In some cases. the reactive system looks more like the sort of planning systems that we will investigate in Chapter 5 in that it manipulates a representation of its pending tasks imposing ordering constraints and dealing with certain classes of interactions between tasks [11]. In others cases, the system is realized as a boolean circuit [8. 26] or as a network of processes that communicate using a specialized message passing protocol [23]. The process of compiling reactive systems from a behavioral specifications is of particular interest. and we will return to this issue in Chapter 5. 8? 10?

4.7 Navigation and Control

Traditionally, the problem of navigation. involving spatial and geometrical modeling, and the problem of control. involving kinematics and dynamical modeling have been considered separately. The former is believed to be in the realm of planning; the latter in the realm of control. In the first problem. we are given a geometrical model describing a robot, the objects surrounding it. their current relative positions and orientations. and some goal state describing a final position of the robot. and we are asked to generate a trajectory or path through the associated space of possible configurations of the robot and the surrounding objects. In the second problem. we are given a dynamical model of the robot. and asked to generate a feedback control law that issues torques to manipulator joints and drive wheels in order to track a supplied reference trajectory. In this section. we consider a unified approach that addresses both of these problems.

To represent the state of the robot with respect to its environment, we introduce the idea of *configuration space* taken from Mechanics and adapted for use in robotics [22]. Following Latombe [20], we represent the robot, $\mathcal{A}$, and the objects—we will refer them as *obstacles*—in its environment, $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m$, as closed subsets of the *work space*, $\mathcal{W} = \mathbf{R}^n$, where $n = 2$ or 3. Both the robot and the obstacles in the workspace are assumed to be rigid. Let $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{W}$ be Cartesian frames of reference embedded in $\mathcal{A}$ and $\mathcal{W}$ respectively. $\mathcal{F}_\mathcal{A}$ is a moving frame while $\mathcal{F}_\mathcal{W}$ is fixed.

A *configuration*, q, of an object is a specification of the position and orientation of $\mathcal{F}_\mathcal{A}$ with respect to $\mathcal{F}_\mathcal{W}$. The *configuration space*, $\mathcal{C}$, is the set of all configurations of $\mathcal{A}$. We employ the Euclidean metric and the following distance function to induce a topology on $\mathcal{C}$. The distance between two configurations, $q, q' \in \mathcal{C}$, is defined as

$$\mathtt{distance}(q, q') = \max_{a \in \mathcal{A}} \|a(q) - a(q')\|,$$

where $\|x - x'\|$ denotes the Euclidean distance between any two points, $x, x' \in \mathbf{R}^n$, and $a(q)$ is the point in $\mathcal{W}$ occupied by $a \in \mathcal{A}$ when $\mathcal{A}$ is in configuration q. We define the *free space*, $\mathcal{C}_{free}$, to be

$$\mathcal{C}_{free} = \{q | q \in \mathcal{C} \wedge \mathcal{A}(q) \cap (\bigcup_{i=1}^{m} \mathcal{B}_i) = \emptyset\},$$

where $\mathcal{A}(q)$ is that subset of $\mathcal{W}$ occupied by $\mathcal{A}$ in configuration q. A *free path* (or just a *path*) of $\mathcal{A}$ from some initial configuration, q, to the *goal* configuration, q^*, is a continuous map

$$\pi : [0, 1] \rightarrow \mathcal{C}_{free},$$

subject to the constraints that $\pi(0) = q$ and $\pi(1) = q^*$.

The literature is full of approaches to solving the problem of finding obstacle-free paths in configuration space. In the following, we consider the *artificial potential field* approach first introduced to the robotics community by Khatib [17] which unifies navigation (or path planning) and control. Our treatment here borrows the notation of Latombe [20], as well as some of the insights of Koditschek [19] on the connections between planning and control. To simplify the subsequent discussion, we assume that the robot is a point object and the workspace is $\mathbf{R}^2$. In this case, it is meaningless to talk about the robot's orientation, and, hence, the configuration space is identical to the work space.

We wish to design an artificial potential field so that the robot will be attracted toward the goal configuration in C and repulsed by obstacles. This field of forces is modeled as a function, F, defined by

$$F(q) = -\nabla U(q),$$

where $U : C_{free} \rightarrow \mathbf{R}$ is a differentiable potential function, and the gradient, ∇, is defined in the case of $C = \mathbf{R}^2$ as

$$\nabla U = \begin{bmatrix} \partial U/\partial x \\ \partial U/\partial y \end{bmatrix}.$$

We represent the potential function as a sum of attractive and repulsive component potential functions:

$$U(q) = U_{att}(q) + U_{rep}(q).$$

Generally, the attractive force is represented either as a conic potential well using the Euclidean distance, as in

$$U_{att}(q) = \xi \|q - q^*\|,$$

where ξ is a positive scaling factor, or as a parabolic potential well using the Euclidean distance squared, as in

$$U_{att}(q) = \frac{1}{2}\xi \|q - q^*\|^2,$$

where the constant 1/2 is just to make ∇ come out a little neater. In the former case, we have

$$\nabla U_{att}(q) = \xi \frac{(q - q^*)}{\|q - q^*\|},$$

and in the latter

$$\nabla U_{att}(q) = \xi (q - q^*).$$

There are advantages and disadvantages to both approaches to representing the attractive potential. In some cases, it is useful to define a hybrid potential using a parabolic potential within some fixed radius of the goal (facilitating gradient descent search in the proximity of the goal) and a conic potential outside that radius (keeping the potential value smaller at points far from the goal) [20].

We decompose the repulsive component of the potential function into m additive components, one for each obstacle. In designing a repulsive field for a particular obstacle, we want to make it impossible for the robot to come in contact with the surface of the obstacle while allowing movement to proceed unimpeded when the robot is sufficiently distant from the obstacle. For a convex object, $\mathcal{B}_i$, the following potential function performs well

$$U_{\mathcal{B}_i}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{\rho_i(q)} - \frac{1}{\zeta}\right)^2 & \text{if } \rho_i(q) \le \zeta \\ 0 & \text{if } \rho_i(q) > \zeta \end{cases} .$$

where ζ is a positive scalar called the *distance of influence*, and ρ_i is defined as

$$\rho_i(q) = \min_{q' \in \mathcal{B}_i} \|q - q'\|,$$

where we do not bother to distinguish between the configuration space and the work space, since in the cases considered here they are the same.

The gradient of $U_{\mathcal{B}_i}$ is defined by

$$\nabla U_{\mathcal{B}_i}(q) = \begin{cases} \eta\left(\frac{1}{\rho_i(q)} - \frac{1}{\zeta}\right)\frac{1}{\rho_i^2(q)}\nabla\rho_i(q) & \text{if } \rho_i(q) \le \zeta \\ 0 & \text{if } \rho_i(q) > \zeta \end{cases} ,$$

where $\nabla\rho_i(q)$ is defined as follows. Let q_c be the unique configuration in $\mathcal{B}_i$ such that $\|q - q_c\| = \rho_i(q)$. $\nabla\rho_i(q)$ is the unit vector pointing away from $\mathcal{B}_i$ in the direction determined by the line passing through q and q_c.

We combine the repulsive fields for the set of obstacles, $\{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m\}$, by taking a simple sum,

$$U_{rep}(q) = \sum_{i=1}^{m} U_{\mathcal{B}_i}(q).$$

The gradient of the sum is simply the sum of the gradients,

$$\nabla U_{rep} = -\sum_{i=1}^{m} \nabla U_{\mathcal{B}_i}(q).$$

Combining the attractive and repulsive force fields, we have

$$F(q) = \nabla U_{att} + \nabla U_{rep}.$$

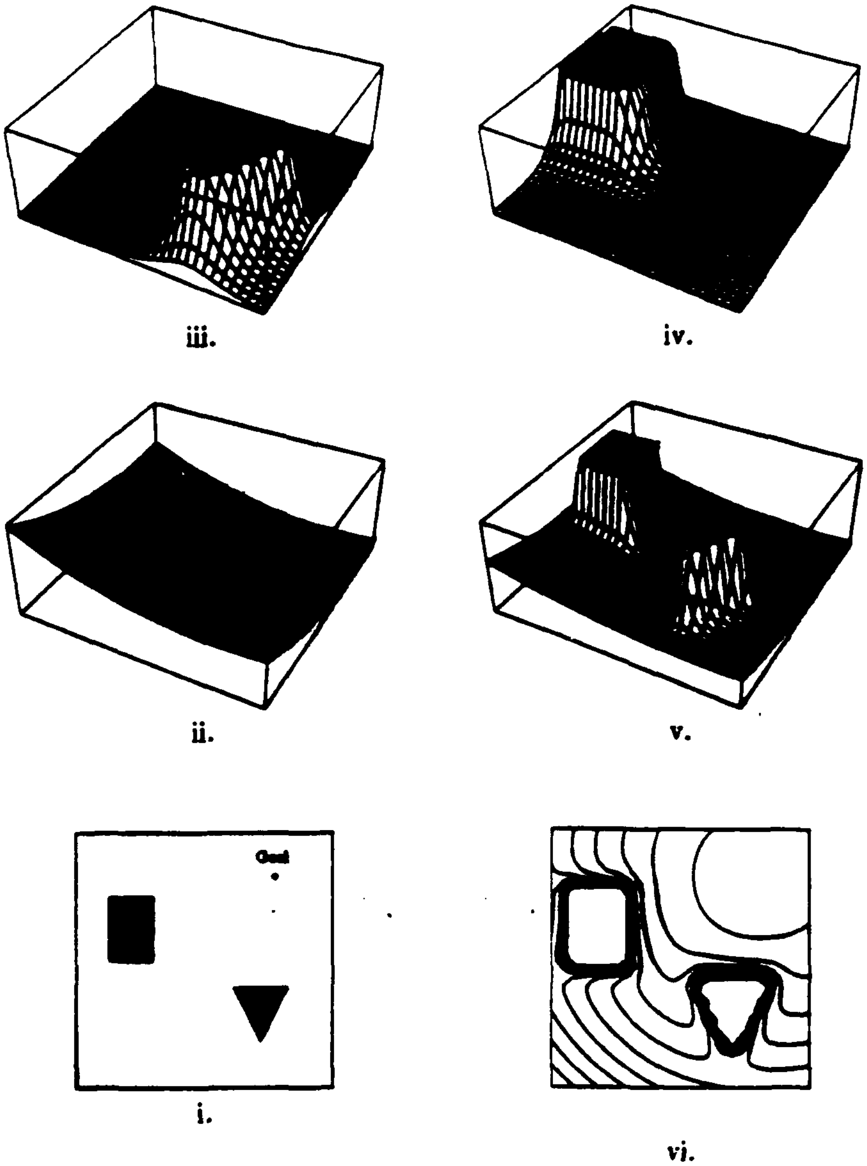


Figure 4.25: A 2-D configuration space (i) containing two obstacles. The attractive potential field (ii) along with the repulsive potential fields (iii) and (iv) for each of the two obstacles, the sum (v) of the attractive and repulsive potential fields, and a 2-D plot (vi) showing several equipotential contours.

Figure 4.25 shows a 2-D configuration space. the resulting potential fields. and several equipotential contours indicating that the potential field has a single minimum. The attractive potential is modeled as a parabolic potential well.

The potential field approach was originally conceived of as a method for real-time obstacle avoidance. The basic idea was to regard the robot in configuration space as a particle moving under the influence of the field. $F = -\nabla U$. The acceleration is determined by $F(q)$ for every $q \in \mathcal{C}$. Given the dynamics of $\mathcal{A}$ and assuming perfect sensing and motors that deliver exact and unlimited torque. we can compute the torques that should be issued to each of the actuators so that the robot behaves exactly as the particle metaphor predicts.

Consider a very simple robot with ~~a~~ single degree of freedom (*e.g.*. a prismatic (sliding) joint). We assume that its position (configuration). $q \in \mathcal{C} = \mathbf{R}$. and velocity. $\dot{q}$. can be measured precisely by a perfect sensor and controlled by a servo that delivers exact and unlimited force, $\mathcal{F}$. We model the dynamical system using Newton's second law of motion,

$$M\ddot{q} = \mathcal{F},$$

where M is the mass of the robot. The object is to move the robot from its present configuration to some final configuration q^*.

In the potential field approach described above. we address the geometrical side of the problem in terms of optimizing a cost function disguised as a potential function. This approach is quite similar to the dynamic programming example that we investigated in Section 4.5. The cost function that we are trying to minimize in this case is just the attractive potential function introduced earlier

$$\varphi = \frac{1}{2}K_P\|q - q^*\|^2.$$

where K_P is any positive scalar. To simplify the present discussion. we ignore the problem of avoiding obstacles. From this equation. we obtain

$$\dot{q} = -\nabla\varphi = -K_P(q - q^*),$$

and note that, since in this case q^* is the only minimum of φ. this linear differential equation generates a solution to the geometric problem of finding a path from any initial starting configuration to q^*. Now we set out to derive a control law that will serve to track the path (or reference trajectory) so defined.

Having interpreted φ in terms of potential energy, we define the kinetic energy, κ, as

$$\kappa = \frac{1}{2} M \dot{q}^2,$$

and obtain the total energy, λ, as the difference of the kinetic and potential energies

$$\lambda = \kappa - \varphi.$$

A dynamical model can be obtained using the Lagrangian formulation of Newton's equations defined by

$$\frac{d}{dt}\left(\frac{\partial \lambda}{\partial \dot{q}}\right) - \frac{\partial \lambda}{\partial q} = \mathcal{F}_{ext},$$

where $\mathcal{F}_{ext}$ represents all of the external (non-conservative) forces acting on the robot. The resulting Newtonian law of motion is

$$M\ddot{q} - K_P(q - q^*) = \mathcal{F}_{ext}.$$

Let us assume that $\mathcal{F}_{ext}$ represents a dissipative force (we can add this if necessary) proportional to the velocity,

$$\mathcal{F}_{ext} = -K_D \dot{q},$$

where K_D is a positive scalar. The resulting system is asymptotically stable, and converges to the goal q^* from all initial configurations $q \in C$.

Finally, we have

$$M\ddot{q} + K_D\dot{q} - K_P(q - q^*) = 0.$$

Returning to our original dynamical model

$$M\ddot{q} = \mathcal{F},$$

we can obtain the following control law

$$\mathcal{F} = -K_D\dot{q} + K_P(q - q^*),$$

an instance of proportional derivative feedback control. The proportional component captures the essence of a simple one-dimensional planning system that determines an appropriate reference trajectory in configuration space. The derivative component enables the controller to respond appropriately to the behavior of the two-dimensional (one spatial and one temporal dimension) physical system.

Khatib's motivation for employing artificial potential fields was to provide real-time obstacle avoidance capability for multi-link manipulators [17]. In his original formulation, it was assumed that there would exist a higher level of control that would compute a global strategy in terms of intermediate goals. The low-level system would produce the necessary forces to achieve these goals, accounting for the detailed geometry, kinematics, and dynamics in real time. In the following, we say a bit more about the high-level problem of computing a global strategy corresponding to a path from the current configuration to the goal configuration.

The approach to building potential fields described earlier has a number of problems: some of which can be easily remedied and others of which are more difficult to overcome. We address some of these problems now, beginning with the easiest ones, working our way up to the more difficult.

The repulsive field for obstacles in the workspace was defined only for convex objects. We can extend the method to handle more general objects by decomposing each obstacle into some number of (possibly overlapping) convex objects, associating a repulsive potential with each component, and summing the result. There are some subtlies with this approach (see [20]), but this basic method of decomposition works well in practice.

The next problem concerns the assumptions regarding the dimensions of the workspace and the degrees of freedom of the robot. For the idealized point robot operating in two dimensions, the two-dimensional configuration space was equivalent to the Euclidean plane. In general, the number of parameters required to describe the configuration of the robot will determine the dimension of the configuration space. For a rigid robot operating in three dimensions, it takes six parameters to describe the configuration of the robot. For manipulators consisting of rigid links serially connected by single-degree-of-freedom joints (*e.g.*, revolute (rotating) and prismatic (sliding) joints), the number of parameters required is equal to the number of joints. For existing mobile robots and manipulators, it is possible to construct the requisite configuration spaces and extend the techniques described above to handle the resulting motion planning problems. However, assuming P $\neq$ NP, the complexity of planning free paths is exponential in the dimension of the configuration space.

In general, computing free paths for multi-link manipulators and mobile robots in cluttered environments can be quite expensive [27]. From the perspective of computational complexity, this high-level geometric planning

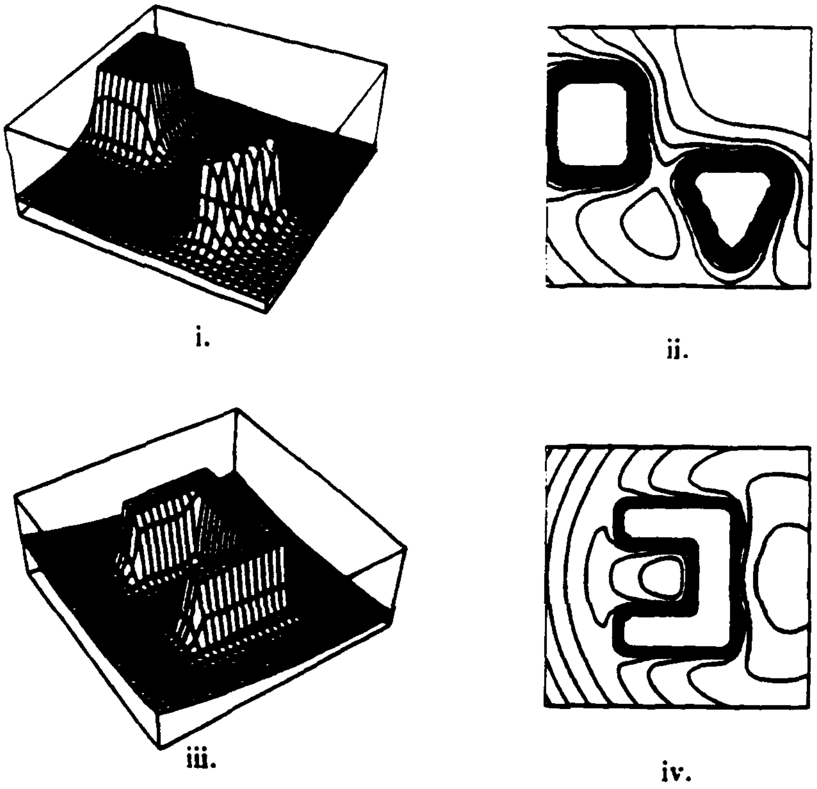


Figure 4.26: Two potential fields with multiple extrema: one (i) resulting from two closely situated convex obstacles, and a second (iii) resulting from a single concave obstacle. A set of corresponding equipotential contours is shown (ii) and (iv) for each of the two potential fields.

problem is typical of the sort of problems that we will encounter in the next chapter. Solutions to problems involving a significant number of constraints (*e.g.*, an environment cluttered with obstacles) and many alternative control actions (*e.g.*, robots with several degrees of freedom) tend to be computationally prohibitive. For real-time applications involving such problems, it is generally necessary to make simplifying assumptions thereby decreasing the complexity of the resulting decision problem while at the same time sacrificing generality and possibly risking soundness or completeness.

Another problem with the artificial potential function approach outlined earlier concerns with the problem of multiple extrema in potential fields. In general, a potential field for a cluttered work space may include several extrema. Under such conditions, using the gradient to guide search may result in paths that terminate at extrema other than the one corresponding to the goal configuration. Concave objects are one potential source of misleading local extrema (see Figure 4.26.iii), but such extrema can also result in the case of closely situated convex obstacles if the distance of influence, ζ, is greater than twice the distance between the obstacles (see Figure 4.26.i).

In order to avoid falling into local minima, it is necessary to employ more sophisticated search methods than simple gradient descent. In the following, we consider one such method for finding collision-free paths in a two-dimensional configuration space.[7]

We begin by tessellating the configuration space to form a grid of equally sized cells. In the case of a point robot on a planar surface, the discretized configuration space, $\mathcal{C}_{\mathbf{Z}}$, is a subset of the integer plane, $\mathbf{Z} \times \mathbf{Z}$:

$$\mathcal{C}_{\mathbf{Z}} = \{(i,j) | 0 \leq i, j \leq r\},$$

where r is a integer parameter used to bound the size of the configuration space. The potential at the coordinates, (i,j), in the integer plane is $U(il, jl)$ where l is the length of the side of a cell. We assume that both the initial and the goal configurations are configurations in $\mathcal{C}_{\mathbf{Z}}$, and that, if two configurations are neighbors in $\mathcal{C}_{\mathbf{Z}}$ and both of them belong to $\mathcal{C}_{free}$, then the straight line segment connecting them also lies in $\mathcal{C}_{free}$.

In the following, T is a tree whose nodes are configurations in $\mathcal{C}_{\mathbf{Z}}$. We define a ***best-first path planning algorithm*** as follows.

1. Initialize T to be the tree consisting of the single (root) node corresponding to the current configuration.

2. Choose a leaf node, q, of T with unexplored neighbors in $\mathcal{C}_{\mathbf{Z}}$ whose potential value is equal to or less than the potential value of all the other leaves in T with unexplored neighbors.

3. Add to T as children of q all configurations not already in T whose potential value is less than some (large) threshold. (This threshold is set to avoid paths that get too close to obstacles. Recall that at the surfaces of obstacles the potential is infinite.)

[7] The method for searching two-dimensional configuration space described here can be extended to higher-dimensional configuration spaces with little modification, but is only practical for dimension $\leq$ 4 [20].

4. If q^* is a leaf node in T, then go to Step 6.

5. If there are no leaf nodes in T with unexplored neighbors, then return failure, else go to Step 2.

6. Return the path from the root of T to q^*.

The algorithm described above is guaranteed to find a free path if one exists or report failure otherwise. The algorithm deals with multiple extrema by following a discrete approximation to gradient descent until reaching a local minimum. Once in a local minimum, it proceeds to "fill in" the well of this minimum by exploring the surrounding cells until a saddle point is reached and the local minimum is avoided. By adding simple optimizations to facilitate finding the next node to explore, it is possible to achieve a running time of $O(mr^m \log r)$ for a configuration space of dimension m. The algorithm works for configuration spaces of arbitrary dimension, but for dimension much greater than four the running time is prohibitive.

It should be noted that the best-first planning algorithm will find a path if one exists, but not necessarily the shortest path or the optimal path by any given metric. The discretized configuration space can be used as part of a dynamic programming approach to finding optimal paths. Indeed, using a dynamic programming approach, we can design an algorithm that will construct a potential field with a single minima at q^* in $O(mr^m)$. Using this potential field, one can generate the shortest path from any initial location to q^* using a discrete approximation to gradient descent in time linear in the length of the path.

Koditschek [18] provides a method of generating potential functions which he calls *navigation* functions that have a single global minimum. The advantage is that simple local methods (*e.g.*, gradient descent) suffice for navigation and control. However, as with other approaches to motion planning, the cost of generating navigation functions can be quite high in the case of cluttered environments and robots with many degrees of freedom.

that

This section was meant as a bridge between the central issues of this chapter and those of the next. In this chapter, we considered basic properties of dynamical systems such as controllability, observability, and stability that are critical in the design of control systems. We investigated the fundamental idea of feedback control and considered the use of performance measures in optimal control. Finally, in this section, we considered the idea of providing higher-level direction for control in the context of navigation problems. In particular, we considered methods for encoding navigation tasks in terms of potential functions that provide a convenient basis for the control of manipulators and mobile robots. The next chapter considers the issues involved in encoding high-level tasks in much more detail. Like the problems involved in motion planning, the problems we will be looking at in the next chapter are computationally complex.

4.8 Further Reading

The literature on control systems theory and practice is vast. In the following, we point out some books and articles that have been particularly useful in understanding the basic control issues and their attendant mathematical formulations. For a good overview of classical and modern approaches to control, the introductory text by Dorf [10] is excellent. Most control texts assume a relatively high level of mathematical sophistication. In particular, some familiarity with linear systems analysis is generally assumed. The text by Chen [9] provides a good introduction to linear systems theory. Gopal's book [14] on the control of linear multivariable systems is an excellent introduction to that subject. For more of an engineering perspective on control, the interested reader is advised to consult Bollinger [5] or Borrie [6].

The survey article by Ramadge and Wonham [25] provides a good introduction to work in the area of discrete events systems. Optimal control texts generally rely on a good background in the differential and integral calculus, and, in particular, the calculus of variations [12]. Athans and Falb [2] provide an introduction to optimal control. There have been many books written on dynamic programming. The original text by Bellman [3] is still generally available and provides a good introduction to the subject with plenty of illustrative examples.

For a careful treatment of the configuration space representation and a variety of approaches to finding free paths in configuration space, the reader is encouraged to read Latombe's book on robot motion planning [20]. Koditschek [19] provides a technical and historical survey of navigation techniques using potential functions including a discussion of stability issues. For a survey of complexity results pertaining to motion planning, see Schwartz, Sharir, and Hopcroft [28].

Bibliography

[1] Aho, Alfred V.. Hopcroft. John E.. and Ullman. Jeffrey D.. *Data Structures and Algorithms.* (Addison-Wesley, Reading, Massachusetts. 1983).

[2] Athans, Michael and Falb. Peter L.. *Optimal Control: An Introduction to the Theory and Its Applications.* (McGraw-Hill, New York, 1966).

[3] Bellman. Richard. *Dynamic Programming.* (Princeton University Press. 1957).

[4] Bellman, Richard, *Adaptive Control Processes,* (Princeton University Press, Princeton, New Jersey, 1961).

[5] Bollinger, John G. and Duffie, Neil A., *Computer Control of Machines and Processes,* (Addison-Wesley, Reading, Massachusetts, 1988).

[6] Borrie, John A.. *Modern Control Systems: A Manual of Design Methods.* (Prentice-Hall. Englewood Cliffs, New Jersey, 1986).

[7] Brooks, Rodney A.. A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation,* 2 (1986) 14–23.

[8] Chapman, David and Agre. Philip E., Pengi: An Implementation of a Theory of Activity, *Proceedings AAAI-87. Seattle, Washington,* AAAI, 1987, 268–272.

[9] Chen, C. T., *Introduction to Linear System Theory,* (Holt, Rinehart. and Winston, New York. 1970).

[10] Dorf. Richard C.. *Modern Control Systems.* (Addison-Wesley. Reading, Massachusetts, 1989).

[11] Firby. R. James. An Investigation in Reactive Planning in Complex Domains. *Proceedings AAAI-87. Seattle. Washington.* AAAI. 1987. 196–201.

[12] Gelfand. I. M. and Fomin. S. V.. *Calculus of Variations.* (Prentice-Hall. Englewood Cliffs. New Jersey. 1963).

[13] Georgeff. Michael P. and Lansky. Amy L.. Reactive Reasoning and Planning. *Proceedings AAAI-87. Seattle. Washington.* AAAI. 1987. 677–682.

[14] Gopal. M.. *Modern Control System Theory,* (Halsted Press. New York. 1985).

[15] Hayes-Roth. Barbara. A Blackboard Architecture for Control. *Artificial Intelligence.* **26** (1985) 251–321.

[16] Kalman. R. E.. Falb. P. L.. and Arbib. M. A.. *Topics in Mathematical System Theory.* (McGraw-Hill. New York. 1969).

[17] Khatib. Oussama, Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research.* **5** (1986) 90–99.

[18] Koditschek. D., Exact Robot Navigation by Means of Potential Functions: Some Topological Considerations. *IEEE International Conference on Robotics and Automation. Raleigh, NC.* 1987. 1 6.

[19] Koditschek. D.. Robot Planning and Control Via Potential Functions. Khatib. Oussama. Craig. John H., and Lozano-Pérez. Tomás. (Eds.). *Robotics Review 1.* (MIT Press. Cambridge. Massachusetts. 1989). 349–367.

[20] Latombe. Jean-Claude. *Robot Motion Planning,* (Kluwer Academic Publishers. Boston, Massachusetts, 1990).

[21] Lewis. Frank L.. *Optimal Control.* (John Wiley and Sons. New York. 1986).

[22] Lozano-Pérez. Tomás. Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers.* **32** (1983) 108–120.

[23] Nilsson. Nils J.. Action Networks. Tenenberg. Josh. Weber. Jay. and Allen. James. (Eds.). *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*. 1989. 36–68.

[24] Pontryagin. L. S.. Boltyanskii. V. G.. Gamkrelidze. R. V.. and Msichenko. E. F.. *The Mathematical Theory of Optimal Processes*. (John Wiley and Sons. New York. 1962).

[25] Ramadge. Peter and Wonham. Murray. The Control of Discrete Event Systems. *Proceedings of the IEEE*. 77(1) (1989) 81–98.

[26] Rosenschein. Stan and Kaelbling. Leslie Pack. The Synthesis of Digital Machines with Provable Epistemic Properties. Halpern. Joseph Y.. (Ed.). *Theoretical Aspects of Reasoning About Knowledge. Proceedings of the 1986 Conference*. Los Altos. California. Morgan-Kaufmann. 1987. 83–98.

[27] Schwartz. J. T. and Sharir. M.. On the Pianos Movers' Problem: I. *Communications on Pure and Applied Mathematics*. **36** (1983) 345–398.

[28] Schwartz. J. T.. Sharir. M.. and Hopcroft. J.. *Planning. Geometry. and Complexity of Robot Motion*. (Ablex. Norwood. New Jersey. 1987).

[29] Wolovich. William A.. *Linear Multivariable Systems*. (Springer-Verlag. New York. 1974).

Chapter 5

Knowledge-Based Planning

Control theory provides a framework for constructing strategies to control processes modeled as dynamic systems. Sometimes, however, it is more convenient to represent the controlled process in terms of causal event models of the sort investigated in Chapter 3.[1] The problem of constructing courses of action based on properties of causal event models is called *planning*, and the specification for intended actions of the robot over time is called a *plan*. By planning, the robot in effect programs itself to act in a particular way in the future. AI researchers have developed a variety of planning techniques, applicable for a wide assortment of plan and event representations.

In the general planning setup, the robot is given a causal event model, with a distinguished subset of events, called *actions*, deemed under the robot's control. In other words, the robot can directly establish the truth of actions, but can influence other events only indirectly through their causal relations to actions. The robot also has some *objectives* describing desirable properties of the controlled process in terms of patterns of events. Planning is the process of assembling basic actions into a composite plan object designed to further these objectives.

A large fraction of planning effort is typically devoted to reasoning about the effects, or potential consequences, of actions. One important reasoning task is to determine whether a particular property should be expected to hold at some point after or during the plan's execution. Planners perform this task by applying their *truth criterion* to the causal event model. The

[1] *It would be nice to provide some suggestions about what features of the process indicate the best choice of representation. Potential advantages of event-based (linguistic) ontology include facilities for representing incomplete information and the intuitive appeal of causal events. Perhaps a comparative discussion belongs at the start or end of Chapter 3.*

computational expense of determir'ng which propositions hold at various points in time depends strongly o.. the representation for the effects of actions and the accuracy of the algorithm implementing the truth criterion. For the planning techniques described below, we use deducibility with respect to TEMPLOG causal models as the truth criterion.

Usually it is not possible to predict perfectly the effects of actions on the controlled process. These limitations are manifest by indeterminacy or even incorrectness of the truth criterion. To plan effectively under these circumstances, the robot may need to gather information directly from the controlled process, augmenting the predictions drawn from its causal model. This approach is directly analogous to the use of feedback in control systems. In robot planning, the process of sensing the state to influence subsequent action is called *execution monitoring.*

Planning is *deliberative*, in that it generally calls for a broad consideration of the available courses of action and their potential consequences. However, in most situations the robot does not have the luxury of unbounded deliberation, because the process of interest progresses in time as the robot computes its plan. To produce effective action under the stress of real time, the planner must have some capability to *react* to its perceived situation without necessarily invoking its full deliberative powers. For any planning problem there is a spectrum of computational strategies, expected to produce better plans as more time is devoted to deliberation. Managing this tradeoff is a significant issue in the design of comprehensive planning architectures.

The final issue we consider in this chapter concerns the specification and interpretations of the robot's fundamental objectives in control. In the common approaches to planning (including the one we present here), objectives are represented as a set of predicates, called *goals*, on states of the controlled process. The planning task then amounts to finding a course of action guaranteed to achieve these goals. As we have seen in several examples, however, absolute goal conditions cannot express gradations of preference needed to capture the realistic objectives of a control problem. The basic difficulty is that predicates coarsely partition the outcomes into two sets, failing to distinguish among states where the goals are achieved, and providing no guidance whatever for problems where it is impossible to guarantee goal achievement. When objectives can be achieved to varying degrees or with some probability, the more general preference representation is required to properly account for the tradeoffs inherent in choosing alternative courses of action. On the other hand, the goal representation meshes well with the event ontology for causal modeling, and with plan evaluation procedures based on the truth criterion. Moreover, goals have significant heuristic value in focusing the search for good plans, and therefore constitute a useful approximation for more expressive preference structures. At the end of this chapter, we analyze the preferential interpretation of goals as a first step toward a reconciliation of common planning practice with the general theories of decision and control.

5.1 A Task Reduction Approach

The approach to planning we describe here is organized around the concept of a *task*, which is an abstract operation that the robot is committed to performing. Tasks are abstract in the sense that they dictate the general nature of what the operation is to accomplish without necessarily specifying its precise implementation. Before an abstract task can be carried out, the planner must supply sufficient detail so that it can be executed directly by the robot hardware.

One way of increasing detail is to replace an abstract task with a more specific task or collection of more specific tasks. This process of refining the level of abstraction is called *task reduction*. Upon *reducing* an abstract task, the robot commits to carrying out the more specific tasks. The reduction process continues until all the tasks are specified in sufficient detail or all avenues of reduction are exhausted.

A task detailed enough to be executed by robot hardware is called *primitive*. Of course, primitiveness is a relative property, defined with respect to the capabilities of a particular execution module. For complex planning problems, it is often useful to construct a hierarchy of abstraction levels, each corresponding to a virtual robot with its own set of actions that are considered primitive. In this scheme, the planner at each level generates tasks at the next lowest level of detail, but is viewed as an execution module by the level immediately above.

One important type of nonprimitive task[2] comprises those committing the robot to make a given proposition hold. These *achievement* tasks are denoted achieve(P), where P is the particular proposition to be achieved. Such tasks may be reduced by finding a primitive task that necessarily achieves P, or by finding some other tasks achieving propositions that collectively entail P.

[2] *Other types include maintenance and prevention. Mention these, but do not introduce them into the logic. Perhaps give interpretation for them in terms of achieve.*

Reduction is complicated by the fact that a* any instant the robot is likely to have many tasks, and several methods fo. reducing any given one. In other words, finding a method to achieve a proposition is a search problem. It is quite possible that a choice for reducing one task may preclude potential reductions for some of the other tasks, requiring backtracking. Sometimes these conflicts can be detected and avoided, by coordinating the reduction of separate tasks via constraints. In the remainder of this section, we present a scheme for task reduction, developing a set of data structures and associated techniques for organizing and managing the search process.

As far as our temporal model is concerned, a task is just a special sort of time token. An instance of a task is created by asserting an expression of the form

`token(task(`*type*`),`*symbol*`)`.

The assertion declares that the robot has a task of type *type* throughout the interval from `begin(`*symbol*`)` to `end(`*symbol*`)`. For instance, the following expressions assert that the robot has two particular tasks, one primitive and the other an achievement.

```
task(push_button(button42)).
task(achieve(location(robot,valve1))).
```

Primitive tasks are specified by their type. If the query `primitive(`Q`)` succeeds, then Q is the type of a task that can be directly executed on robot hardware. In the warehouse domain, we assume that `push_button(`B`)` is primitive,[3] where B is the label of a known push-button control switch. Being primitive does not imply that executing an action will necessarily achieve the proposition of the achievement task it was reduced from. The intended results are typically guaranteed only under certain conditions, which may or may not be entirely under the robot's control.

Tasks come and go as the robot discovers information about its environment. If the robot enters the loading area and notices a truck that was not there the last time it visited, then it will formulate a new task to load that truck. Conversely, if the robot currently has the task to load `truck45`, and it notices that `truck45` is no longer waiting, the robot will give up on this task. To institute the general policy of servicing trucks waiting in the loading area, we assert a task covering that policy, and add a projection rule to the database relating this task to its more specific instances.

[3] *Comment from Jean-Claude Latombe that this can actually be a complex operation from the robot control perspective.*

```
project(task(service_trucks),
        becomes(location(Truck,loading_dock)),
        task(load(Truck))).
```

along with a corresponding policy to give up on load tasks when they are no longer feasible.[4]

```
project(task(load(Truck)),
        becomes(¬location(Truck,loading_dock)),
        ¬task(load(Truck))).
```

Of course, for the above polices to work as intended, the robot has to be continually aware of new arrivals and unexpected departures, and, hence, it might be reasonable to have policies that call for the robot to occasionally scan the loading area looking for changes. This points out a problem with our representation of time and action; we do not distinguish between what is true of the world and what the robot knows to be true of the world. We return to this issue in Section 5.2.

Some policies should be ignored in certain situations. For instance, whenever the robot is in an area where an assembly operation is in progress, it should check to see if the assembler's malfunction light is on, and, if so, generate a task to push the reset button. However, if the robot is in a hurry or has only recently checked the malfunction light, it might not generate the task to check. The decision whether or not to check will depend upon what other tasks the robot currently has pending.

Some types of policies are more difficult to administer than others. For instance, a policy to clean up concrete spills might generate a specific task in response to each detected spill, but what about a policy to prevent or minimize concrete spills? In the latter case, the robot's response to a predicted spill might simply be to change its current plan by, say, opening an input valve a little less or an output valve a little more, but the robot might instead decide that the valve settings are perfect and choose to prevent spillage by raising the walls of the mixing tank. Whether this latter approach is acceptable will depend upon the cost of raising the walls. In Section 5.4, we consider how more precise specifications of objectives, in the form of value functions, may provide the information necessary for such decisions.

In the task reduction approach, planning knowledge is encoded in expressions of the form

todo(*what*, *when*, *how*).

[4] *What if the load task is already reduced? Presents complicated problem of how to maintain status of tasks in reduction search.*

where *what* is a task type, *when* is an interval, and *how* is either another task type or a compound task description specifying how to reduce the *what* task type. If *how* is a simple task, the result of interpreting the todo expression is to introduce a new task

```
token(task(how),when).
```

and mark the original *what* task as "reduced," to note that we need not search for another method.

One common *how* task type is the no_op, or do-nothing action. In general, when you have a task to accomplish something that is already true, the obvious action to perform is none at all. We can represent this simple strategy as:

```
todo(achieve(P),K,no_op) :- holds(end(K),P).
```

where, in order to absolve the robot of its commitment to achieve P, all that is important is that P is true at the end of the interval K.

Note that providing methods for achievement tasks in todo expressions significantly simplifies the search process. Without these methods, the planner would have to examine the causal model directly to find controllable events that would result in the proposition to be achieved. By relying on them, however, the robot will not in general consider every possible way of accomplishing its task. The task reduction approach implicitly assumes that the computational benefits of using todo directives exceeds the cost of supplying them and the loss of opportunities potentially derived from a direct analysis of the causal model.

It is often useful to group together a collection of tasks coordinated for a common purpose. We call the description of such composite action a *plan*. Actually, these plan objects only partially specify the full course of action, and we sometimes emphasize this by calling them *abstract* or *partial plans*. In contrast, a complete plan is comprised entirely of primitive actions with a precise specification of the time that each is to be executed.

In our task reduction scheme, a plan consists of a set of steps with associated constraints that determine their order and duration. For instance, a plan to fill a tank might include the following tasks as steps:

```
Step1: achieve(location(truck42,loading_dock))
Step2: achieve(location(robot,valve1))
Step3: achieve(position(valve1) = 35°)
Step4: achieve(floor(robot,floor1))
Step5: achieve(location(robot,valve2))
```

along with constraints on those steps as follows:

```
end(Step1) ⪯ begin(Step2)
distance(begin(Step2),end(Step2)) ∈ [00:00,00:01]
```

The steps in a plan are transformed into a set of tokens in the course of formulating a specific instance of that plan. For example, the above steps might be instantiated as

```
token(task(achieve(location(truck42,loading_dock))),step141).
token(task(achieve(location(robot,valve1))),step142).
token(task(achieve(position(valve1) = 35°)),step143).
token(task(achieve(floor(robot,floor1))),step144).
token(task(achieve(location(robot,valve2))),step145).
```

and then constrained temporally by instantiating the specified constraints:

```
end(step141) ⪯ begin(step142).
distance(begin(step142)),end(step142)) ∈ [0,00:01].
```

where step141 through step145 are newly minted symbols identifying the intervals associated with the task instances.

Plans are represented in our scheme by expressions of the form

plan(*steps*, *time-constraints*, *protections*)

where the *steps* indicate the new tasks involved in the reduction, the *time-constraints* restrict the order of those tasks, and the *protections* specify special properties that must be maintained during the plan's execution. The new tasks are referred to as *subtasks* of the task they were reduced from, inversely designated the *supertask* of the new tasks. All subtasks are implicitly constrained to occur during the interval of the supertask, as specified in the todo expression. Protections are important in detecting problems that arise when one task interferes with another.[5] Consider the following general method for making two propositions true at the same time:

```
todo(achieve((P,Q)),K,
     plan([achieve(P),achieve(Q)],
          [end(1) ⪯ end(K),end(2) ⪯ end(K)],
          [protect(end(1),end(K),P),
           protect(end(2),end(K),Q)])).
```

[5] *Latombe: what about simplifications possible by merging identical subtasks for different supertasks?*

The steps are numbered by their position in the list of steps.[6] The constraints refer to these numbers and are used to constrain the corresponding tokens created in the process of instantiating a particular plan. The two protections stipulate that to achieve the conjunction of P and Q, achieve each of P and Q individually, and ensure that once each proposition is made true it remains so at least until the end of time interval K. A protection is said to be *violated* when the robot becomes committed to an action with an effect whose type contradicts the type of the protection.[7] Certain combinations of tasks can make it impossible to avoid violating protections.[8] In some cases, conflicts among propositions to achieve are easy to detect, for instance:[9]

```
achieve((status(assembler,on),status(assembler,off)))
```

In general, however, the interactions between tasks can be arbitrarily complex, requiring considerable effort to detect and resolve.

Most of the plans for a given application encode domain-specific strategies for reducing abstract tasks to more concrete ones. The set of all such strategies constitutes a *plan library*. In the following, we provide examples of plans that might appear in the plan library for a robot operating in the warehouse domain. We take the liberty of simplifying the plans somewhat (*e.g.*, by leaving out certain steps and constraints) in order to make the text more readable. Here is a plan for installing an option in an appliance:

```
todo(achieve(installed(Option,Appliance)),K,
     plan([achieve(location(Appliance,in_conveyor)),
           achieve(location(Option,in_conveyor)),
           achieve(status(assembler,on))],
          [end(1)≼begin(3),end(2)≼begin(3)],
          [protect(end(1),begin(3),
                   location(Appliance,in_conveyor)),
           protect(end(2),begin(3),
                   location(Option,in_conveyor)),
           protect(end(3),end(K),status(assembler,on))])) ←
    holds(begin(K),(status(assembler,off),
                    status(malfunction_light,off))).
```

[6] *Explain how this might be implemented in prolog using a pre-processor, and how the syntax might be further sugared to use step identifiers.*

[7] *Latombe: What about temporary violations? Is there any way to allow them? Answer: never really useful; consider modal truth criterion.*

[8] e.g., *the Sussman anomaly.*

[9] *Make clear that conflicts are not errors but can represent legitimate competition among goals and subgoals.*

Take note of the role protections play in this plan. The first two protections ensure that, once placed on the assembler's input conveyor, the appliance and the option to be installed will remain there until the robot starts the assembly. The third protection prevents the robot from inadvertently scheduling some other activity that would result in turning the assembler off during its execution of the installation task.

The robot will also need plans for changing the location of objects. The following general rule specifies how to change the location of something other than the robot:

```
todo(achieve(location(Object,Loc1)),K,
     plan([achieve(location(robot,Loc2)),pick_up(Object),
           achieve(location(robot,Loc1)),set_down(Object)],
          [end(1)≼begin(2),
           end(2)≼begin(3),
           end(3)≼begin(4)],
          [protect(end(1),begin(2),location(robot,Loc2)),
           protect(end(2),begin(3),holding(robot,Object)),
           protect(end(3),begin(4),location(robot,Loc1))])) ←
    holds(begin(K),(location(Object,Loc2),
                    Object≠robot,Loc1≠Loc2)).
```

The above plan assumes a somewhat implausible model of robotic movement. In order to move an appliance onto the input conveyor, the robot would have to move itself onto the conveyor while holding the appliance, then set the appliance down so that it rests on the conveyor. Although we continue to make use of such simplifications as required to keep the discussion focused, we return to consider continuously changing parameters in general and spatial inference in particular later in this chapter.[10] To plan for moving the robot about, we use the following rule, and assume that the task type move(*source*, *destination*) is primitive:

```
todo(achieve(location(robot,Loc1)),K,move(Loc2,Loc1)) ←
    holds(begin(K),location(robot,Loc2)).
```

Finally, the robot needs a plan for turning the assembler on or off:

```
todo(achieve(status(assembler,Stat1)),K,
     plan([achieve(location(robot,assembly_area)),
           push_button(Stat1)],
          [end(1)≼begin(2)],
          [protect(end(1),begin(2),
                   location(robot,assembly_area))])) ←
    holds(end(K),(status(assembler,Stat2),Stat1≠Stat2)).
```

[10] *Will we? I don't think so.*

Now we are ready to consider how to go about reducing a set of abstract tasks to primitive tasks. In general, the reduction process can be quite complex. We start by sketching an algorithm for performing the reduction, give an example illustrating the algorithm in operation, and then comment on complications not explicitly handled by the algorithm. The task reduction procedure is specified as follows:

1. Find some task, token(task(*what*),*when*), which is neither primitive nor marked as already reduced. If no such task exists, wait until a new task is added to the database.

2. Using the query, todo(*what*,*when*,*how*), try to find some method *how* for carrying out the task found in Step 1.

3. If the query specified in Step 2 fails, try adding constraints to restrict the ordering of the existing tasks. This may trigger rules permitting the todo query to succeed on the next attempt.

4. If the query specified in Step 2 fails even after trying various additional constraints, try removing one or more of the existing tasks along with all associated protections and other constraints. Be careful to reinstate the original supertask.

5. If Step 2 through Step 4 fail to produce an applicable method, return to Step 1 and try another task.

6. If the query succeeded, mark the original task as reduced and add the new *how* task or plan to the database, along with any specified constraints and protections.

7. Upon effecting the reduction, TEMPLOG will have updated the database using the projection and persistence clipping algorithm, and the projection rules that describe the effects of the actions. Check to see if any protections are violated by the addition of the new tasks.

8. If any protections are violated, resolve the violation by either reordering or removing one or more of the existing tasks.

9. Go to Step 1.

A concrete example should help illustrate the basic operation of the reduction algorithm. Figure 5.1 shows a TEMPLOG database containing one

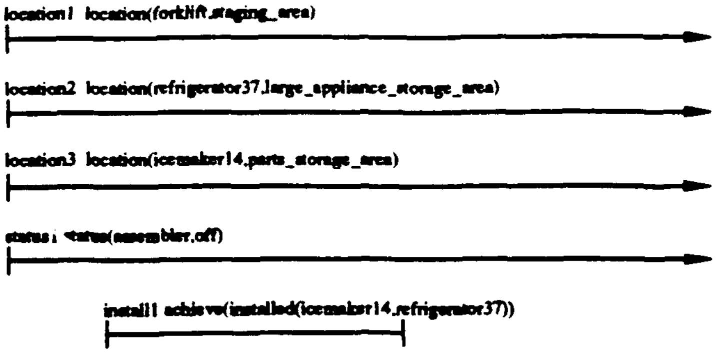

location1 location(forklift,staging_area)
location2 location(refrigerator37,large_appliance_storage_area)
location3 location(icemaker14,parts_storage_area)
status1 status(assembler,off)
install1 achieve(installed(icemaker14,refrigerator37))

Figure 5.1: Database before reduction

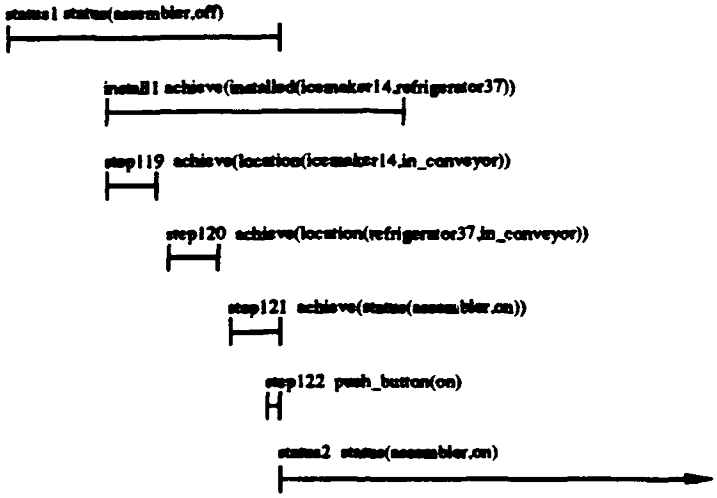

status1 status(assembler,off)
install1 achieve(installed(icemaker14,refrigerator37))
step119 achieve(location(icemaker14,in_conveyor))
step120 achieve(location(refrigerator37,in_conveyor))
step121 achieve(status(assembler,on))
step122 push_button(on)
status2 status(assembler,on)

Figure 5.2: Database after reduction

nonprimitive unreduced task to install an ice maker in a refrigerator. Figure 5.2 shows the TEMPLOG database resulting from applying the reduction algorithm, using the planning knowledge specified in this section and the knowledge of cause-and-effect relationships described in Chapter 2. (Only selected steps are depicted in Figure 5.2 to keep the display readable.) The reduction illustrated in Figure 5.2 is a particularly simple one; we consider next some problems that may arise in more complicated situations.

Returning to the previous listing of the reduction algorithm, note that there are a number of steps where choices are made. In Step 1, the robot will generally have to choose from a number of unreduced nonprimitive tasks. In Step 2, there are likely to be several methods for reducing the chosen task. If the todo query does not immediately succeed, the robot may have to consider several alternative orderings in Step 3, or several reduced sets of tasks in Step 4, before it is able to find a reduction strategy that works. In fact, the iteration of Steps 1 through 5 can cause the algorithm to loop indefinitely, continually removing tasks and adding new ones. In general, the algorithm is not guaranteed to eventually terminate with a complete reduction. The problem of resolving protection violations in Step 9 can be particularly troublesome; sometimes involving numerous attempts at reordering or modifying the set of tasks. If the robot makes the wrong choice early in the planning process, it may expend a great deal of effort before it "backs up" and tries an alternative option. All of these problems and more have to be routinely solved by a robot control system that generates plans by task reduction. Researchers have developed an array of techniques for dealing with these problems, although none offer a complete solution.

For an example of how the procedure detects and resolves negative interactions among tasks, suppose that the TEMPLOG database depicted in Figure 5.1 also contains a task committing the robot to perform routine service on the assembler. Suppose further that this routine service task is currently scheduled to overlap with the task to install the ice maker in the refrigerator. The plan for routine-service tasks is specified below:

```
todo(routine_service(assembler),K,
     plan([achieve(status(assembler,off)),
            lubricate(assembler),
            replenish_coolant(assembler),
            push_button(reset)],
           [end(1)≼begin(2],end(1)≼begin(3),
            end(2)≼begin(4),end(3)≼begin(4)],
           [protect(end(1),begin(4),
                   status(assembler,off))]).
```

Note that the routine service plan requires that the assembler be turned off before the lubrication and coolant-replacement tasks are initiated. The task to turn the assembler off conflicts with the installation plan, which requires that the assembler be on.

Figure 5.3 depicts the database resulting from reducing both the installation and routine service tasks. Note that the database predicts that the assembler will not remain on throughout the required portion of the installation interval. In the course of reducing the two tasks, the robot should have generated two protections, the first associated with the installation task:

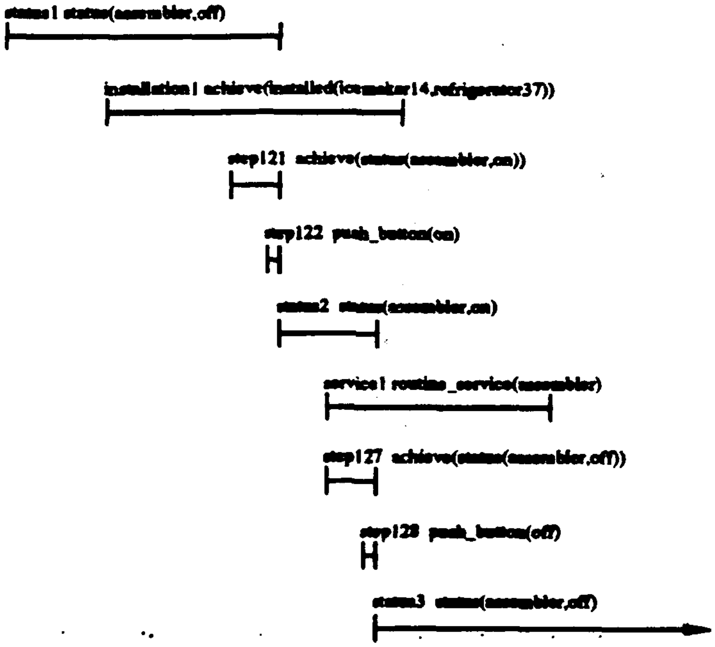

status1 status(assembler,off)
installation1 achieve(installed(icemaker14,refrigerator37))
step121 achieve(status(assembler,on))
step122 push_button(on)
status2 status(assembler,on)
service1 routine_service(assembler)
step127 achieve(status(assembler,off))
step128 push_button(off)
status3 status(assembler,off)

Figure 5.3: Database with a protection violation

```
protect(end(step121),end(installation1),status(assembler,on))
```

and the second associated with the routine service task:

```
protect(end(step127),end(service1),status(assembler,off))
```

These two protections conflict with one another (*i.e.*, they require the persistence of tokens of contradictory types over a common subinterval). The easiest way to resolve this particular conflict between the installation task and the routine-service task is to reorder the two tasks: either constrain the interval service1 to end before the beginning of step127, or constrain service1 to begin after installation1. For other conflicts, reordering may not suffice, necessitating more drastic measures.

There are other problems that can arise besides protection violations. Many of the rules specifying reduction methods have conditions that must hold if the reduction method is to apply. We refer to these conditions as *reduction assumptions*. For instance, consider the general rule for avoiding unnecessary work:

```
todo(achieve(P),K,no_op) ← holds(end(K),P).
```

If the robot has a task of type achieve(status(assembler,off)) during token interval81, when the assembler is already expected to be off, then it will reduce the task to a no_op. The reduction assumption is that status(assembler,off) holds at end(interval81). The robot will check at reduction time that the reduction assumption holds, but the assumption may become false during subsequent planning as additional tasks are added to the database. Reduction assumptions have to be carefully monitored in much the same way that protections are, and steps taken when the assumptions are found to be violated.[11]

The general problem of reducing a set of tasks to primitive tasks so as to avoid violating any protections or falsifying any reduction assumptions is believed to be computationally intractable (*i.e.*, it has been shown to be in the class of *NP-hard* problems). Deadlines and reasoning about resources are obvious sources of complexity, but, even if we were to ignore deadlines and resources, most interesting planning problems remain in the company of those difficult problems. For certain versions of the problem, there is no

[11] ***If the reduction trigger needs to hold at task time, why aren't these always protected? Or alternately, why not allow protections with simple task reductions? Clarify the utility of defining the concept of reduction assumptions distinct from protections. Confusing factor: protections seem to guard against inter-task conflicts as a side effect of preventing intra-task conflicts, performing some of the function of reduction assumptions.***

effective method for generating plans (*i.e.*, the problem is undecidable). For the problems that are decidable, it is fairly simple to write an algorithm that finds a solution if one exists, and signals that no solution exists otherwise. Unfortunately, such an algorithm may take an unacceptably long time to return its answer. While these observations are somewhat discouraging, we at least know that good approximate solutions are possible (*e.g.*, humans perform reasonably well driving forklifts in warehouses). In artificial intelligence, planning problems are typically recast as search problems, and standard methods have been applied to develop heuristic algorithms that perform well in practice. In this chapter, we have not explored the various search techniques, concentrating instead on the basic problem of how a robot might use symbolic representations to guide its behavior.

In each iteration of the reduction algorithm, a partially completed plan is analyzed and modified. For some planning problems, such incremental analysis is problematic. The projection rule describing the process of moving from one location to another (specified in Chapter 2) indicates that the distance in time between when the move is initiated and when the robot is in the final location is a function of the distance in space between the robot's initial and final position. This rule brings up an important issue that we have avoided so far. The order in which tasks are executed determines to a large extent how long they take to execute. If the robot is trying to minimize the time spent in execution or avoid violating deadlines, then it has to consider not only the order in which to perform each task, but the location that it has to be in to perform each task and how to travel between those locations. Task scheduling with deadlines and travel time inevitably involves nasty combinatorics and NP-hard problems.

There are all sorts of deadlines that a robot might have to contend with in practice. In addition to absolute deadlines (*e.g.*, finish before noon), there are graded deadlines (*e.g.*, the longer you take, the more it will cost you), and relative deadlines (*e.g.*, finish before the tub overflows). The last are particularly interesting from the perspective of control. How do you coordinate the behavior of a robot with that of other processes over which the robot has only partial or intermittent control? We have already mentioned how one might accomplish such coordination for the tank-filling problem using feedback. In the following, we consider how we might accomplish the necessary coordination using planning, for a somewhat more complex problem.

Recall the problem presented in Chapter 1 involving a robot in a concrete plant scurrying about from one valve to another trying to fill trucks with

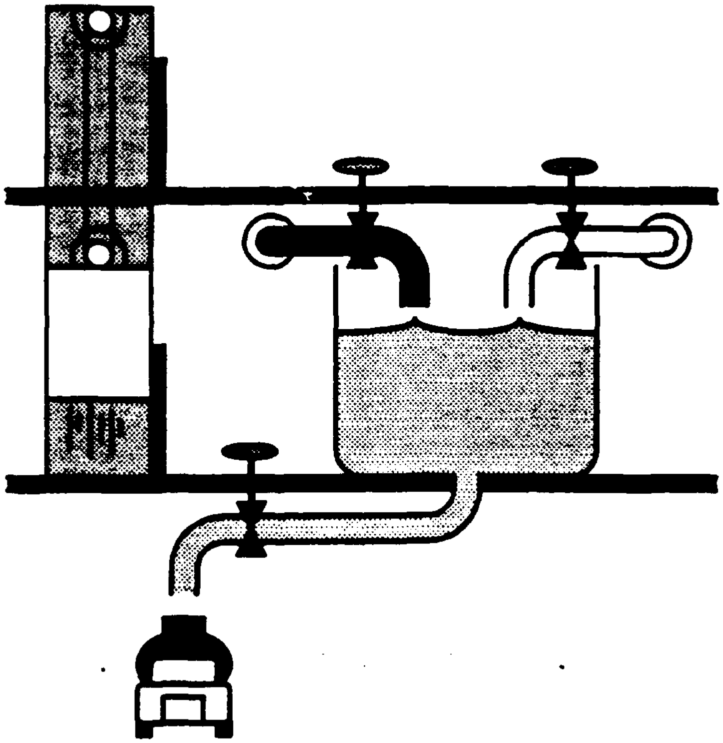


Figure 5.4: The concrete factory domain

```
todo(achieve(full(Truck)),K,
     plan([achieve(location(Truck,loading_dock)),
            achieve(position(valve(in1)) = 35°),
            achieve(position(valve(in2)) = 35°),
            achieve(position(valve(out1)) = 35°),
            achieve(position(valve(out1)) = 0°),
            achieve(position(valve(in2)) = 0°),
            achieve(position(valve(in1)) = 0°)],
           [end(1)⪯ begin(2)),
            distance(begin(2),end(2)) ∈ [00:01,00:02],
            distance(end(2),begin(3)) ∈ [00:01,00:02],
            distance(begin(3),end(3)) ∈ [00:01,00:02],
            distance(end(3),begin(4)) ∈ [00:01,00:02],
            distance(begin(4),end(4)) ∈ [00:01,00:02],
            distance(end(4),begin(5)) ∈ [00:14,00:16],
            distance(begin(5),end(5)) ∈ [00:01,00:02],
            distance(end(5),begin(6)) ∈ [00:01,00:02],
            distance(begin(6),end(6)) ∈ [00:01,00:02],
            distance(end(6),begin(7)) ∈ [00:01,00:02],
            distance(begin(7),end(7)) ∈ [00:01,00:02]]) ←
    holds(begin(K),(0° ≤ position(valve(in1)) ≤ 5°,
                    0° ≤ position(valve(in2)) ≤ 5°,
                    1.5m ≤ fluid_height(tank14) ≤ 2.0m,
                    25m³ ≤ tank_size(Truck) ≤ 35m³)).
```

Figure 5.5: A plan for filling a single truck

properly mixed concrete. Figure 5.4 depicts the basic layout of the concrete factory.

The simplest approach is to provide a small number of canned solutions, each covering a subset of the situations that the robot might find itself in. For instance, Figure 5.5 shows a plan for filling a single truck. If the tasks are carried out within the specified time constraints, then this plan guarantees that no concrete is spilled, the two ingredients, cement and aggregate, are mixed in the proper proportions (*i.e.*, 50/50 give or take 5%), and that the tank is filled to at least 90% of its capacity. To achieve the required degree of coordination, the tasks are tightly constrained with respect to one another. Figuring out how the individual tasks are achieved will require further reduction. If the robot is to carry out all of the tasks itself, it will have to move between the various valve locations (or *stations*) and perform the indicated valve adjustments in the times allotted. The plan for changing the position of a valve is simply:

```
todo(achieve(position(Valve) = Theta),K,
     plan([achieve(location(robot,station(Valve))),
           turn(Valve,Theta)],
          [end(1) ≼ begin(2)],
          [protect(end(1),end(2),
                   location(robot,station(Valve)))])).
```

The process of turning a valve is modeled by the following projection rule, which bounds the time it takes for the turning to complete.

```
project(position(Valve) = Theta1,
        turn(Valve,Theta2),
        [(|Theta1-Theta2| ÷ max_turning_speed),
         (|Theta1-Theta2| ÷ min_turning_speed)],
        position(Valve) = Theta2).
```

Moving from one location to another is complicated by the fact that the stations for the input and output valves are located on different floors. We assume that there are two ways of going from one floor to another: by elevator or stairs. When it is in service, using the elevator is always preferred to taking the stairs.

```
todo(achieve(floor(robot,Floor1)),K,
     use_elevator(Floor1,Floor2)) ←
    holds(begin(K),(status(elevator,in_service),
                    floor(robot,Floor2),Floor2 ≠ Floor1)).

todo(achieve(floor(robot,Floor1)),K,
     use_stairs(Floor1,Floor2)) ←
    holds(begin(K),(not(status(elevator,in_service)),
                    status(stairs,in_service),
                    floor(robot,Floor2),Floor2 ≠ Floor1)).
```

The plans for using the elevator and stairs are straightforward.

```
todo(use_elevator(Floor1,Floor2),K,
     plan([achieve(location(robot,elevator_landing(Floor1))),
           achieve(floor(elevator_cab,Floor1)),
           achieve(location(robot,elevator_cab))],
          [end(1)⪯begin(3),end(2)⪯begin(3)])).
```

```
todo(use_stairs(Floor1,Floor2),K,
     plan([achieve(location(robot,stair_landing(Floor1))),
           negotiate_stairs(Floor1,Floor2)],
          [end(1)⪯begin(2)])).
```

where we assume that negotiating the stairs is primitive:

```
project(location(robot,Floor1),
        negotiate_stairs(Floor1,Floor2),[00:03,00:05],
        location(robot,Floor2)).
```

and the elevator begins to operate as soon as the robot enters the cab:

```
project(floor(elevator_cab,Floor),
        becomes(location(robot,elevator_cab)),[00:01,00:02],
        location(robot,other(Floor))).
```

Now, suppose that the robot is given the task to fill a particular truck, truck42. The robot's task is indicated by the following token.

```
token(task(achieve(full(truck42))),fill45).
```

State the initial conditions, valve flow factors, tank area and height, truck capacity, and status of stairs and elevator. To avoid introducing plans for summoning the elevator, assume that the elevator, if it is in service, is always on the same floor as the robot. Get material from Dean and Siegle, AAAI-90.

We can reduce fill45 using the plan shown in Figure 5.5 and either the elevator plan or the stairs plan. The reduction using the elevator plan is preferable because it manages to fill the truck three minutes earlier than the reduction using the stairs plan. Although we have provided no mechanism to express this general preference, the relative time requirements are taken into account in reasoning about interactions between competing tasks. For example, suppose that the robot has another task constrained to occur during fill45, which involves running a systems diagnostic program requiring to remain idle for ten minutes. In this case, there is only one solution consistent with the constraints: the reduction using the elevator plan.

There are a number of potential problems with the type of plan shown in Figure 5.5. One arises in trying to apply such plans to coordinate two simultaneous fillings or to orchestrate a series of fillings. It would be necessary in general to provide special plans for each particular filling combination. Another difficulty is that if the flow rate for one of the valves or the volume of the mixing tank changes, then the plan no longer guarantees avoidance of spillage and suitable mixture proportions. For instance, if the flow rate of `valve(in1)` is increased by a factor of 10%, then the reduction using the elevator plan will result in a task duration of 24 minutes, but there will be 2m^3 of concrete spilled on the floor and an unacceptable 2:3 ratio of cement to aggregate in `truck42`.

As an alternative to excessively specific plans, we could provide general plans that do not specify exact valve positions and task durations, and hence give up the guarantees regarding results like spillage and mixture. A search algorithm would then heuristically choose positions and durations to use in generating candidate plans, and the candidate satisfying the mixture constraints that provides the least spillage would be chosen for execution. The advantage of such a scheme is its improved prospect for finding a solution over a broad range of task situations. The disadvantage is that the set of all combinations of valve positions and task durations is quite large, only a small subset of which are likely to yield good solutions.[12]

A compromise is to have a small number of highly specific plans that are likely to produce solutions close to satisfying the achievement tasks and then heuristically adjust the plan parameters to improve performance. For example, heuristics might include "if the truck is not filled to 90% of its capacity, then start closing the output valve later" or "if the mixing tank spills over, then open the output valve more and close it earlier." Research in planning tends to focus on general-purpose domain-independent methods. It is important to remember, however, that the performance of a particular planning system can be dramatically enhanced by bodies of special-purpose knowledge encoded in the form of domain-dependent rules.

One important issue that we avoided in the previous examples involves the representation of plans in which an action is repeated some number of times. For instance, how do you represent a plan to unload a truck containing seven appliances? Using the list manipulation routines in PROLOG, this turns out to be relatively easy. A more difficult problem involves planning to unload a truck with some unknown number of appliances. We would like to be able to predict the type of the subtasks involved and how long the unloading is likely to take. We might specify a recursive plan such as:

[12] *What about parameterized plans, where the precise settings are specified as a function of the other variables? This is a form of conditional plan, to be discussed in next section.*

```
todo(achieve(empty(Truck)),K
     plan([unload_item(Truck),achieve(empty(Truck))],
          [end(1)≼begin(2)])) ←
    holds(end(K),¬empty(Truck)).
```

This gives us an idea of the types of subtasks involved, but we cannot determine their number because it does not make sense to reduce the recursive (second) step until after some item is unloaded. Thus, we are still left with the problem of estimating how long the unloading task will take. We could estimate how many items are likely to be on a given truck, and expand a plan with this number of subtasks. This remains short of a complete reduction, as we cannot determine where the robot will have to travel until we know the exact contents of the truck.

A more general problem with the sort of approach specified above is that it relies on *execution-time replanning*. Because the effects of the plan are not completely predictable, the subsequent course of action cannot be specified until after the results are known, at which time the task reduction process is resumed. The drawback of this strategy is that task reduction involves deliberate search, and thus may entail a considerable pause in the robot's constructive activity. This pattern of alternation between planning and execution can waste through idleness a considerable fraction of the robot's resources. Worse, the continuing evolution of the controlled process during deliberation may erode or eliminate the robot's opportunity to effectively promote its objectives.

One way to address this problem is to provide, at plan time, for alternate courses of action depending on conditions holding at execution time. In the following, we consider methods for constructing and reasoning about plans that explicitly refer to such contingencies. These plans include knowledge acquisition steps to collect information, associated with alternative subplans to be performed or not, conditional on the information gained during plan execution.

5.2 Conditional Plans

Faced with the task to unload a particular truck with unknown cargo, there are (at least) two approaches. The robot might construct a plan to find out what appliances are on the truck, and postpone planning their removal until the contents are known. Alternatively, it might create a plan that includes a step to determine the appliances in need of unloading, plus some additional steps conditional upon the outcome of the initial information-gathering operation. This second approach produces a *conditional plan*, and has a number of advantages over postponing planning entirely. For instance, while the robot may not know exactly what appliances are on the truck, it does know that in order to move them it will need a screwdriver to remove the restraining straps that protect them from damage in transit. The plan to unload the truck will require a step to remove the restraining straps no matter what appliances are on the truck. If the robot is currently near a tool box, it can save itself a trip by appending a task to fetch a screwdriver to the beginning of the plan to unload the truck.

More importantly, the conditional plan provides the robot with the means to commit to an answer conditional upon information gathered at execution time. Given a conditional plan, the robot can avoid reinvoking the planner upon determining the contents, and can proceed immediately with the unloading plan specified for the situation actually encountered. However, this readiness is achieved only at the price of computing contingency plans for unloading all potential types of cargo. As all but one of these plans goes unused, there is a considerable computational overhead in generating the contingency plans. This is the fundamental tradeoff in generating conditional plans, an issue we discuss further in Section 5.3. In this section, we present some simple mechanisms for expressing and reasoning about conditional action.

To specify conditional actions in plans, we introduce a new task type:

cp(*condition*, *conditional_action*, *alternate_action*)

If the condition holds at task execution time (*i.e.*, the interval specified in its task token), then the robot *is* to perform the conditional action; otherwise it is to perform the alternate action. To illustrate the use of cp, consider the following method for moving to a particular floor. The plan is to use the elevator if it is in service, otherwise to take the stairs.

```
todo(achieve(floor(robot,Floor1)),K,
     cp(status(elevator,in_service),
        use_elevator(Floor1,Floor2),
        use_stairs(Floor1,Floor2))) ←
   holds(begin(K),(location(robot,Floor2),Floor1 ≠ Floor2)).
```

Note that although it includes no temporal argument, the conditional expression implicitly refers to the status of the elevator during K, the interval in which the tasks are operative. Recall that in reducing tasks using todo, the new task (in this case, conditional) inherits the interval of the original task.

The appropriate application of a cp method relies on two assumptions. First, it makes sense to introduce a conditional task only if the value of the condition *is not already known* at the time of introduction. In the example above, this means that the robot cannot determine at planning time whether the elevator will be in service during K. We can verify this assumption by augmenting the rule's antecedent:

```
    holds(begin(K),(location(robot,Floor2),Floor1 ≠ Floor2,
                              not(status(elevator,in_service)))).
```

We rely here on negation as failure to satisfy the query in cases where the elevator status at begin(K) cannot be determined. Having modified the rule, we should also add to the plan library an *unconditional* todo method for the case where the elevator is known to be in service during K.

The second assumption underlying conditionalization is that the value of the condition *will be known* at the time of task execution. This prerequisite is much more difficult to ensure. Suppose we implement the conditionalization using a pair of projection rules:

```
project(task(cp(Cond,Act,_)),becomes(Cond),task(Act)).
project(task(cp(Cond,_,Act)),becomes(¬Cond),task(Act)).
```

The problem with this approach is that we have no assurance that either Cond or ¬Cond will become true during the interval of interest. Moreover, it confuses what is true in the model with what the robot knows to be true. We can alter the syntax all too easily.

```
project(task(cp(Cond,Act,_)),becomes(knows(Cond)),task(Act)).
project(task(cp(Cond,_,Act)),becomes(knows(¬Cond)),task(Act)).
```

Unfortunately, it is not at all straightforward to define expressions of the form knows(φ) in a manner consistent with both our intuitions about the meaning of knowledge and the behavior of our temporal logic. Instead, we present a simpler approach based on explicit declarations of the observability of events. Although this scheme does not provide for complicated inferences about the knowledge state of the robot, it covers many useful situations with minimal additional machinery. In Section 5.5, we evaluate the limitations of our observability approach with respect to more general theories of knowledge.

Our first step toward managing the generation of conditional plans is to restrict the class of propositions that are eligible for conditioning. The basic constraint is that the robot can execute a conditional action only if the condition is part of its available information. To impose this constraint, we define a special class of propositions, called *observables*, that comprise the exclusive domain of conditional expressions.

A proposition φ is declared observable during the interval $\langle t_1, t_2\rangle$ by an assertion of the form observable(t_1,t_2,φ).[13] Given this declaration, the planner is permitted to specify cp tasks for proposition φ during subintervals of $\langle t_1, t_2\rangle$.

It is important to distinguish the temporal extent of the observable proposition from the time the robot observes it. For example, the robot might find out at t_1 (when it reads the maintenance schedule) whether the elevator will be in service at some subsequent time t_2. We would express such a situation by asserting:

holds(t_1,observable(t_2,status(elevator,_))).

Observability at a given time has implications for observability at other times. For instance, it is reasonable to postulate that observability is persistent; that is, the robot does not forget:

holds(T1,observable(t,φ)) ← holds(T2,observable(t,φ)),T2$\preceq$T1.

But of course we cannot assume that, just because the robot can observe whether φ holds at t, it can also observe whether φ holds at $t+\epsilon$. In other words, a similar persistence relation does not apply for the temporal extent of the observable proposition.

In the common conditional planning situation, the time of observation and the temporal extent of the observable proposition coincide. Given a conditional task of the form cp(φ,_,_) during interval K, we are most concerned with whether:

holds(begin(K),end(K),observable(begin(K),end(K),φ)).

It is precisely this fact that determines whether the cp task is executable by the robot. If the robot is committed to a conditional plan, therefore, it follows that it should be committed to making the condition observable. We might encode this automatic commitment as a projection rule.

```
project(true,becomes(task(cp(P,_,_))),
               task(achieve(observable(P)))).
```

[13] The use of a proposition as an argument to observable is a syntactic variant, similar to constructs like clips, holds, and others introduced in Chapter 3. As for those predicates, we adopt the usual syntactic conventions in specifying its temporal arguments as either points or intervals. Moreover, we sometimes omit the temporal argument when its value is implicit in the context (*e.g.*, within a task assertion).

The problem of ensuring the executability of conditional plans thus reduces to achieving the necessary observability prerequisites. While this is a difficult problem in general, there are typically a wide range of propositions that are rendered directly observable by primitive actions. Let us call such propositions *testable*, and assume that the query testable(P) succeeds if and only if there exists a primitive action, indexed by test(P), that tests for the proposition P. We therefore have:

```
todo(achieve(observable(P)),K,test(P))← testable(P).
```

We could enforce observability syntactically by requiring that all propositions appearing in conditional expressions be potentially testable. This approach is not as restrictive as it might sound, since we can always push off the complexity to reasoning about the relation of directly testable propositions to properties more central to the robot's planning decisions. Nevertheless, such indirection may be unnatural, and it is often possible—albeit more complicated—to achieve observability of useful conditions by means of explicit planning. In allowing more complex information-gathering behavior, we gain flexibility at the expense of sacrificing the guarantee that all conditional tasks will be executable. For completeness, we note that the meaning of a task that conditions on an unobservable proposition is simply that of the no_op action.

To illustrate some of the potential difficulties involved in reasoning about information-gathering, consider the following plan to determine the level of fluid in a truck sitting in the loading dock at a particular point in time.

```
todo(achieve(observable(end(K),fluid_level(Truck))),K,
     plan([achieve(location(robot,station(meter17))),
           read(meter17)],
          [end(1)⪯begin(2),end(2) = end(K)],
          [protect(end(1),end(2),
                   location(robot,station(meter17)))])) ←
    holds(begin(K),end(K),location(Truck,loading_dock)).

holds(T1,observable(T1,fluid_level(Truck))) ←
    occurs(T1,read(meter17)),
    holds(T1,location(Truck,loading_dock)).
```

If the robot has the task to observe the level of the fluid in the truck currently located in the loading dock, then it can do so by positioning itself in the appropriate place to read the fluid-level meter, and invoking the subroutines necessary to read the meter and process the resulting data. Other knowledge acquisition tasks may require significantly more complicated synchronization.

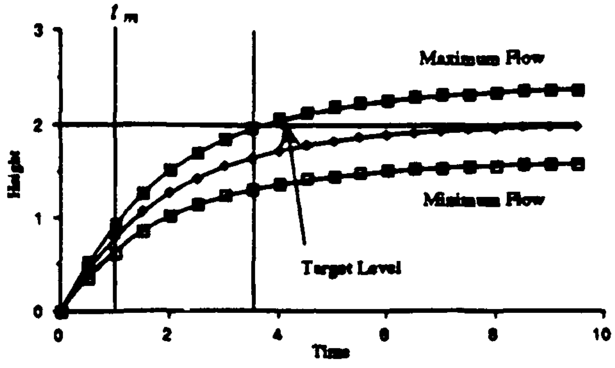


Figure 5.6: Planning with an approximate model

Suppose that the robot wants to close a valve when the fluid level of the truck being filled reaches a particular height. In order to do so, the robot will need to know when the level achieves this height. If the robot lacks a predictive model of the tank-filling process, then it must stand in the appropriate location and monitor the fluid-level meter continuously. If the robot knows the initial conditions and has a precise model of the tank-filling process, then, it can predict exactly when the fluid will reach the target level without consulting the meter at all. If the robot does not know the initial conditions but has a precise model, then it is sufficient that the robot observe the values of the parameters at some point in time in order to predict the height of the tank for all subsequent times. The most likely situation is that the robot will have some estimates for the parameters (perhaps based on measurements at different points in time) and an approximate model whose predictions decrease in accuracy as they extrapolate into the future. Using this information, the robot can generate expectations or worst-case scenarios about when the tank will reach the target level.[14] For instance, suppose that the robot knows the initial conditions for its model at time 0, but its tank-filling model is subject to bounded errors (see Figure 5.6). In planning when to read the meter, the robot must take into account the earliest that the fluid level might reach the target level, as well as the amount of time required to move from the meter to the valve and close it. One possible approach would be for the robot to find its way to the meter at or before the time marked t_m in Figure 5.6, and then replan on the basis of the observed height of the fluid. It could avoid execution-time replanning by indentifying, in advance,

[14] *This is a case of reasoning about the need for feedback, a difficult general problem.*

a threshold on the fluid level upon which it would proceed to the valve. To exhibit maximal robustness, however, the robot must be flexible enough to apply more complex dynamic replanning strategies. For example, if it seems on monitoring for some time that the level is not rising fast enough, the robot might consider opening the valve a bit and rescheduling its subsequent meter readings based on the revised predictions of its flow model.

It should be clear that we could make the dynamic decision problem facing the robot arbitrarily complex. *Use this example to motivate fuller exploration of reactivity in next section. Also point ahead to Chapters 6 and 7, which focus on sensing and reasoning under uncertainty.*

5.3 Planning and Reaction

Discuss in this section, among other things:[15]

- Conditions as the first step toward reactivity. Continuum between unconditional plan languages and universal plans. Conditional plan language defines middle ground.
- Relation of observability approach to control framework.
- Making plans more robust by considering perturbations. Provides for the role of monitoring in plan execution.

Talk about expectations and expectation monitoring during plan execution. What happens when your expectations fail? For example, you try to turn a valve and it doesn't appear to turn or the water level goes up when you close the valve. Talk about replanning and recovering from execution errors. What does it mean to lose, regain, or maintain control? What do you do when things go wrong and you're in the middle of doing something? For instance, the tub is running over and you're on the phone or trying to rescue your dinner from the oven. Develop the analogy between difference-reducing planners and error-driven control strategies.

The idea of reactivity and its contrast to deliberate planning. Architectures for integrating planning methods of the sort discussed in Section 5.1 with reactive systems. Task interpretation systems (see old material). Firby's RAPs. Ties to sections on reactive control in Chapter 4. The "obvious" solution: different levels of competence with varying degrees of reactivity, asynchronous control, run-time arbitration, and off-line compilation for real-time

[15] *Fix transition from preceding section.*

responsiveness. Prelude to architecture for decisi· n-theoretic control of inference, presented in later chapter.

5.4 Goals and Utilities

Limitations of task reduction approach (and classical planning framework) in treatment of goals as predicates. Present more general view of preferences, utility functions, tie to goals, point to decision-theoretic analysis of Chapter 7. How will this be coordinated with the introduction of value functions in Chapter 4?

Paragraph moved from task reduction section. It should also be noted that the reduction planning method described above is not able to handle planning problems in which the criteria for a good plan involve minimizing execution time or maximizing income. While finding a solution that minimizes or maximizes some quan:ity is generally computationally complex, it is still useful to be able to compare candidate solutions. The standard technique for comparing candidate solutions is to use a *value* function to define a metric on the outcomes associated with candidate solutions. The basic idea behind using a value function is simple. Given two candidate solutions (plans), determine the changes over time (referred to as *time lines*) that are predicted to occur as a consequence of executing each plan. The value function is then applied to the resulting time lines and the plan with the lowest cost (highest value) is determined to be the better of the two. Given a set of candidate solutions, one can then select the best. Planning consists of (heuristically) generating a set of candidate solutions, evaluating each candidate, and selecting the best. We discuss this sort of planning in the context of reasoning about deadlines and control.

Generally, choosing an appropriate action requires considering several possible actions and anticipating the consequences of each action. In the case of *PID* control, the designer does all the necessary considering and anticipating at design time and simply encodes his findings in the coefficients of the *PID* controller. This sort of design-time compilation is difficult tc do in general. For instance, finding the shortest tour visiting a set of locations in a factory is a type of problem that might occur frequently for a mobile robot. Computing the solution to even one instance of this type is known to be a hard problem. It would be quite difficult to enumerate and then compute, in advance, the solution to all possible instances of this problem, and, even if you could, it would be difficult if not impossible to store the results of such a prodigious effort on any practical machine.

For any interesting problem, it is impossible or impractical to write down Φ. In the decision sciences, they never even attempt to; rather, they specify belief functions, preferences, and a utility (or value) function. The notion of task is implicit in whatever maximizes expected utility.[16] The introduction of beliefs and expectations is crucial here; what constitutes a task depends critically on a given agent's knowledge, which in turn depends upon what the agent has observed, not just at the last clock tick, but over time, and the agent's ability to reason about those observations. The notion of task in AI is similar despite the fact that the use of value functions is not universally accepted.

Normative vs computational theories of decision-making. The decision sciences provide a "normative" theory of decision making, in that any rational decision maker possessed with the same information and unlimited time to reflect on it would come to the same conclusion. AI, starting with Herb Simon's Nobel-prize-winning model of administrative man, has taken the idea of a resource-bounded agent as a starting point [17].

Motivate need for utility in terms of complications involving Φ. Start with preference order on Ω, then introduce order-preserving, real-valued utility function. Perhaps notation Util is best, by parallel to Val *and given that u and U are already taken in the presentation of control. State the obvious problem with reasoning about elements of Ω and introduce machinery to get around the problem. Introduce a set of time points $\mathcal{T}$, and define time lines in terms of functions from $\mathcal{T}$ to Ω. Redefine Φ accordingly. Introduce the notion of error-driven control laws in terms of a variant on means/ends analysis. If we allow the reference signal to correspond to an arbitrary world state and the controlled variables to include any condition, then the solution to almost any control problem can be characterized in terms of a suitable error-driven control law.*

Explain how goals fit in with this expressive framework. A goal predicate specifies that a state achieving the goal is preferred to one that does not, all else equal. Combining all the expressed goals yields a partial order on states, with preference between competing goals or alternate ways of achieving the same goal not defined. This suggests that goals do not provide sufficient guidance for rational choice of action. Must augment with more precise specification, either by providing strength of preference or finer-grained descriptions of goal predicates and combinations.

[16] *Just as in classical planning it is implicit in what achieves the top-level goal.*

5.5 Further Reading

The material presented on planning is a distillation of a great deal of research. The need for protections was first identified by Sussman [18], and indeed the simplest example of a problem requiring nonlinear plan construction is known as the "Sussman anomaly." The basic idea of reduction interleaved with resolving interactions originated with Sacerdoti's influential NOAH system [14]. Our development of the task reduction approach follows Charniak and McDermott [3], who provide a more comprehensive treatment of protections and search algorithms. The reduction algorithm itself is based loosely on Tate's NONLIN [19] (see Vere [20] for extensions to handle metric time constraints). The notion of policy projection is borrowed from McDermott [10].

Pointers to other work on planning, not necessarily taking task reduction approach. Truth criterion: implicit in much work, made explicit by Chapman. Problems for temporal reasoning about nonlinear plans explored by Dean and Boddy. For a discussion of issues in representing and reasoning about resources, see [22, 5]. General discussions of partial plans (Wellman, Hsu?).

Reasoning about knowledge, action, and perception [4, 9, 11, 12] (especially Morgenstern, Moore). Discussion of observable events and test actions follows Wellman. Evaluate with respect to the more general theories (essentially, the latter allow reasoning about how observability of some facts implies observability of others). General capability for reasoning about knowledge in planning an area of active investigation, with many open questions (see Halpern overview in TARK-86, or survey in Annual Review).

The idea of debugging almost right plans is characteristic of many approaches to planning in artificial intelligence [8, 16, 18].

Reactive planning: AI interest spurred by work of Agre and Chapman [2], Brooks [1], Rosenschein [13], Schoppers [15]. Early example: triangle tables in STRIPS.

Goals and utilities: see our discussion [6], also Haddawy and Hanks [7], Loui, *new paper*.

Bibliography

[1] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

[2] David Chapman and Philip E. Agre. Pengi: An implementation of a theory of activity. In *Proceedings AAAI-87*, pages 268–272. AAAI, 1987.

[3] Eugene Charniak and Drew V. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1985.

[4] Ernest Davis. Inferring ignorance from the locality of visual perception. In *Proceedings AAAI-88*, pages 786–790. AAAI, 1988.

[5] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[6] Thomas Dean and Michael Wellman. On the value of goals. In Josh Tenenberg, Jay Weber, and James Allen, editors, *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*, pages 129–140, 1989.

[7] Peter Haddawy and Steve Hanks. Issues in decision-theoretic planning: Symbolic goals and numeric utilities. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*. DARPA, 1990.

[8] Kris Hammond. Chef: A model of case-based planning. In *Proceedings AAAI-86*, pages 267–271. AAAI, 1986.

[9] Kurt Konolige. *A Deduction Model of Belief*. Pitman Publishing, London, 1986.

[10] Drew V. McDermott. Flexibility and efficiency in a computer program for designing circuits. Technical Report 402, MIT AI Laboratory, 1977.

[11] Robert C. Moore. Reasoning about knowledge and action. Technical Report Technical Note 191, SRI International, 1980.

[12] Leora Morgenstern. A first order theory of planning, knowledge, and action. In Joseph Y. Halpern, editor, *Theoretical Aspects of Reasoning About Knowledge, Proceedings of the 1986 Conference*, pages 83–98, Los Altos, California, 1987. Morgan-Kaufmann.

[13] Stan Rosenschein. Synthesizing information-tracking automata from environment descriptions. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 386–393. Morgan-Kaufmann, Los Altos, California, 1989.

[14] Earl Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, New York, 1977.

[15] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings IJCAI 10*, pages 1039–1046. IJCAI, 1987.

[16] Reid Simmons and Randall Davis. Generate, test and debug: Combining associational rules and causal models. In *Proceedings IJCAI 10*, pages 1071–1078. IJCAI, 1987.

[17] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, Massachusetts, 1981.

[18] Gerald J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975.

[19] Austin Tate. Generating project networks. In *Proceedings IJCAI 5*, pages 888–893. IJCAI, 1977.

[20] Steven Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:246–267, 1983.

[21] Michael P. Wellman. *Formulation of Tradeoffs in Planning Under Uncertainty*. Pitman and Morgan Kaufmann, 1990.

[22] David E. Wilkins. Domain independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–302, 1984.

Chapter 6

Uncertainty in Control

In predicting and controlling the behavior of processes, it is nearly impossible to avoid some degree of uncertainty. Even in cases where an engineer carefully designs a piece of equipment to behave in a particular manner, sources of uncertainty are introduced in manufacturing, in the wear on parts during subsequent use, and through unanticipated interaction with the environment. In this chapter and the next, we consider various approaches to dealing with uncertainty in planning and control. This chapter focuses on uncertainty issues in the context of control systems engineering.

Here, as elsewhere in this book, we make no attempt to provide a comprehensive survey of techniques. Our objective in this chapter is to make several observations about the nature of control as a problem involving uncertainty, and to introduce two techniques that illustrate key issues.

The first technique involves an approach to recovering the state of a dynamical system from observations of its output. The general problem was introduced in Chapter 3 in the discussion of system observability. The solution that we consider here, the Kalman filter, is somewhat specialized, but of broad practical import. In the introduction to a collection of papers on the theory and applications of the Kalman filter, Sorenson [16] writes that, "It is probably not an overstatement to assert that the Kalman filter represents the most widely applied and demonstrably useful result to emerge from the state variable approach of 'modern control theory.' " Our introduction to Kalman filtering emphasizes a basic cycle of activity that is central in the application of the Kalman filtering equations, and is applicable to a wide variety of state estimation problems that do not satisfy the assumptions

[0]©1990 Thomas Dean. All rights reserved.

required for the Kalman filter.

The second technique involves an extension of the dynamic programming approach considered in Chapter 2. The extension is concerned with multistage decision problems in which the dynamical system can be modeled as a stochastic process. We introduce the basic theory in this chapter as it is generally considered as a part of the repertoire of techniques of control. In Chapters 6 and 8, we return to consider the connection between stochastic dynamic programming and various techniques in planning (~~Chapter~~ 6) ~~and learning (Chapter~~ 8). We begin this chapter by considering just how deeply the issues involving uncertainty enter into the problem of controlling dynamical systems. Our treatment here follows that of Koditschek [12].

6.1 Uncertainty and Delay in Dynamical Systems

In both Chapters 2 and 3, we considered a single-degree-of-freedom robot as an example of a simple dynamical system. We continue to resort to such simplified models in this chapter to illustrate our basic points. Let M be the mass of the robot, z its position in some arbitrary frame of reference, and $\mathcal{F}$ the force acting upon the robot. As in Chapter 2, we assume that the plane of motion is horizontal and that there are no frictional forces acting on the robot. The relationship between position, z, and the force, $\mathcal{F}$, is completely determined by Newton's second law of motion.

$$M\ddot{z} = \mathcal{F}$$

The state vector for the dynamical system is defined to be

$$\mathbf{x}(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix},$$

and system state equation is

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 1/M \end{bmatrix} \mathbf{u}(t).$$

In the set-point regulation problem, the task is to transfer the robot from its initial location, $z(t_0)$, to some final (goal) location, z^*, and then keep it there. We begin by giving the controller every advantage in an attempt to avoid the problems introduced by uncertainty. In particular, we assume that the control actuator can exert an arbitrary amount of force, $\mathcal{F}(t)$, at an instant in time, t. We model this using the Dirac delta (impulse) function defined by

$$\int_{-\infty}^{\infty} \delta_\tau(t)\, dt = 1,$$

where

$$\delta_\tau(t) = 0 \ \ \forall t \neq \tau,$$

so that our actuator is able to deliver a pulse of infinite magnitude over an infinitesimally short interval of time possessed of unit area and involving a finite amount of energy.

The controller begins by getting the robot headed in the right direction, namely towards the goal, z^*. We measure the current position and velocity,

$$\mathrm{x}(t_0) = \begin{bmatrix} z(t_0) \\ \dot{z}(t_0) \end{bmatrix},$$

and at the same instant apply an impulse defined by

$$u_{\mathrm{start}}(t) = M(1 - \dot{z}(t_0))\delta_{t_0}(t).$$

The impulse has the effect resetting the initial conditions so that

$$\mathrm{x}(t) = \begin{bmatrix} t + z(t_0) \\ 1 \end{bmatrix} \quad \text{for } t > t_0,$$

and the goal position is achieved at time $t^* = z^* - z(t_0)$. At t^*, we apply a force to exactly cancel the velocity achieved by the first impulse. The second impulse is defined by

$$u_{\mathrm{stop}}(t) = -M\delta_{t^*}(t).$$

The control strategy defined by

$$u(t) = u_{\mathrm{start}}(t) + u_{\mathrm{stop}}(t)$$

provides a solution to our idealized set-point regulation problem. In addition to the assumptions made regarding the Dirac impulse function, this solution relies on the following assumptions.

- We know the exact mass, M, of the robot.
- We can instantaneously and exactly measure the robot's position, z, and velocity, $\dot{z}$.

- We can instantaneously perform all calculations required for control.
- We can exactly measure the elapsed time in order to sequence the velocity canceling impulse.

If any one of the above assumptions fails to hold, then some error will be introduced and this error will become magnified with the passage of time. For instance, suppose that there is some error in the estimate, $\hat{M}$, used for the mass, M. If we apply the same control strategy as before, we obtain

$$\mathbf{x}(t) = \begin{bmatrix} z(t_0) + \left[\frac{\hat{M}}{M} + \left(1 - \frac{\hat{M}}{M}\right)\dot{z}(t_0)\right] t^* + \left(1 - \frac{\hat{M}}{M}\right)\dot{z}(t_0)t \\ \left(1 - \frac{\hat{M}}{M}\right)\dot{z}(t_0) \end{bmatrix} \quad \forall t > t^*,$$

where we have substituted $\hat{M}$ for M in the specification of u_{start} and u_{stop}. From this description of the system state, it should be apparent that small inaccuracies in estimating M will result in finite and *increasing* error in the position of z relative to the goal z^*. Similar errors would occur due to imprecision in measuring the position or velocity at t_0.

This simple example is meant to illustrate how deeply the issue of uncertainty is rooted in the problems of control. Koditschek [12] writes in the same article from which we adapted the above analysis, "The origins of control theory, then, rest in the following observations. Dynamic systems give rise to delay that must be taken into account by any control strategy regardless of available actuator power or sensor accuracy. Moreover, information regarding the real world is inevitably uncertain and may have an adverse effect on performance no matter how small the uncertainty or powerful and accurate the apparatus."

As was pointed out in Chapter ~~8~~ 4, feedback control strategies achieve their robust performance because they continuously account for the error between the measured state of the system and the goal state. Such feedback control systems tend to compensate for measurement and modeling errors. If the measurement and modeling errors systematically mislead the controller, then performance will most certainly be poor; however, feedback controllers often perform well in the presence of certain benign forms of random errors. In the following section, we consider a class of problems for which it is possible to design a module to *estimate* the system state. This module can be coupled to a deterministic feedback regulator to obtain a controller that is optimal by most accepted criteria.

6.2 State Estimation

Suppose that you are designing a system to control the movements of a mobile robot that has to navigate in an office or industrial environment. If you could obtain the exact geometric description for the surfaces of the objects in the surrounding environment, then you could use the planning and control algorithms described in the last section of Chapter 3,[4] or any of a host of other deterministic control strategies to guide the robot on its appointed rounds. Using path planning methods and an exact geometric model for navigation requires that the robot not err in its movement or that the robot correct for errors in movement by reestablishing its position and orientation with respect to the geometric model. This process of reestablishing position and orientation with respect to a geometric model is called *registration* or *localization* in the literature. To help generate a geometric model or maintain registration with an existing model, suppose that the robot has been equipped with a variety of sensors: ultrasonics, infrared, inertial guidance, compass, odometry, laser ranging, tactile sensing. Unfortunately, all of these sensors are prone to errors. In this section, we consider how to design ~~an~~ algorithm[5] that combine~~s~~ (*fuse~~s~~*) the data from all of the sensors, accounting for their tendency to err, ~~so as~~ to provide as accurate a picture of the geometry of the robot's environment as is possible from the data supplied.

The object is to

Consider the following problem in fusing data from different sensors. Suppose we are interested in the distance from the robot to the nearest obstacle surface in the direction the robot is traveling. Sensor 1 reports that the distance is 2 meters, but Sensor 2 reports 5 meters, and Sensor 3 pretty much agrees with Sensor 2, reporting 5.15 meters. The close agreement of two of the sensors would suggest relying on a value close to 5 meters, but it may be that Sensors 2 and 3 are wrong quite often, even systematically wrong, while Sensor 1 is hardly ever wrong. Without additional information about the sensors, it is difficult to know what to do with conflicting evidence. However, if we have prior knowledge about the errors that can be expected from the different sensors, then we may be able to combine the data in a disciplined, perhaps even optimal manner.

In the following, we adopt a Bayesian perspective, and represent our knowledge about sensor errors in terms of conditional probabilities. In particular, if $\mathbf{x} \in \mathbf{R}^n$ represents the system state vector, and $\mathbf{z} \in \mathbf{R}^m$ represents the measurement vector providing information about $\mathbf{x}$, then we represent our knowledge about the performance of the sensors that produced $\mathbf{z}$ as a *conditional probability density function*, $p(\mathbf{x}|\mathbf{z})$, indicating the probabil-

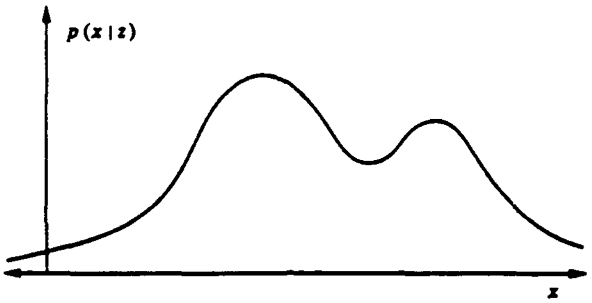


Figure 6.1: The conditional probability density for x given z

ity that $\mathbf{x}$ is the true state of nature given that we have observed $\mathbf{z}$. For a scalar x, the density function might take the form shown in Figure 6.1. More generally, given a discrete dynamical system

$$\begin{aligned} \mathbf{x}(k+1) &= f(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{z}(k) &= h(\mathbf{x}(k)), \end{aligned}$$

where h is a *measurement* function, we will want to calculate a density function of the form

$$p(\mathbf{x}(k)|\mathbf{z}(1), \mathbf{z}(2), \ldots, \mathbf{z}(k)),$$

where $\mathbf{z}(t)$ indicates the measurements made at time t.

Given a conditional probability density function, we wish to determine an estimate of the system state, denoted $\hat{\mathbf{x}}$, to be used for control purposes. Possible candidates for such an estimate are the average or *mean* of the probability distribution corresponding to the density, the peak or *mode* of the distribution, and the *median* of the distribution.[1]

In the following, we assume a linear dynamical system corrupted by "white Gaussian" noise. The assumption that the noise be white requires that the noise value not be correlated in time (*i.e.*, knowledge of the value of the noise at one point in time tells you nothing about the value of the noise at later times).[2] The assumption that the noise be Gaussian requires that

[1] For a scalar quantity, the median is that value of x such that half of the probability mass lies to the left of it and half to the right.

[2] Whiteness also requires that the noise have equal power at all frequencies; a requirement that is impossible to achieve in practice given that all real physical systems respond

the probability density for the amplitude of the noise at any particular point in time take on the familiar bell-shaped curve of a Gaussian distribution.[3]

The assumption of Gaussian noise is often justified by observing that, if the noise is generated by a large number of separate processes, then the sum of their effect can be approximated by a Gaussian distribution. However, the most compelling reason for accepting the assumption of white Gaussian noise is the same as that for accepting the assumption of linearity, namely, it makes the mathematics tractable. As an example of how the Gaussian assumption simplifies things, a Gaussian distribution is completely determined by its first- and second-order statistics, its mean and variance. The Gaussian assumption will also simplify our choice for an estimate of the state given the density; under the assumption of Gaussian noise, the mean, mode, and median all coincide. What is surprising is that, despite the fact that the assumptions seldom if ever are met in dealing with real physical systems, the basic methods that we describe in the sequel have met with extraordinary success in practice [16].

To make our assumptions explicit in the model, we represent the state of the system at time $k+1$ by

$$\mathbf{x}(k+1) = f(\mathbf{x}(k), \mathbf{u}(k)) + \mathbf{v}(k),$$

where f models the response of the dynamical system to a given input, and $\mathbf{v}(k)$ is a vector of zero-mean, white, Gaussian noise processes, modeling the input disturbance or process noise. Let $\mathbf{z}(k)$ represent the (observable) output of the system at time k, so that

$$\mathbf{z}(k) = h(\mathbf{x}(k)) + \mathbf{w}(k),$$

where h models the physics of the measurement process and $\mathbf{w}(k)$ is a vector of zero-mean, white, Gaussian noise processes, modeling the measurement

only within a narrow range of frequencies called the system *bandpass*. For practical purposes, however, the noise will often behave as if white within the bandpass of the system. In certain cases in which the noise is not constant over the system bandpass or is correlated in time, a special "shaping filter" can be added to the system to achieve a model of a dynamical system driven by white noise [14].

[3] The Gaussian or *normal* distribution, $N(\mu, \sigma^2)$, for a (scalar) random variable, x, with mean μ and variance σ^2 (σ denotes the standard deviation) is characterized by the normal probability density:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

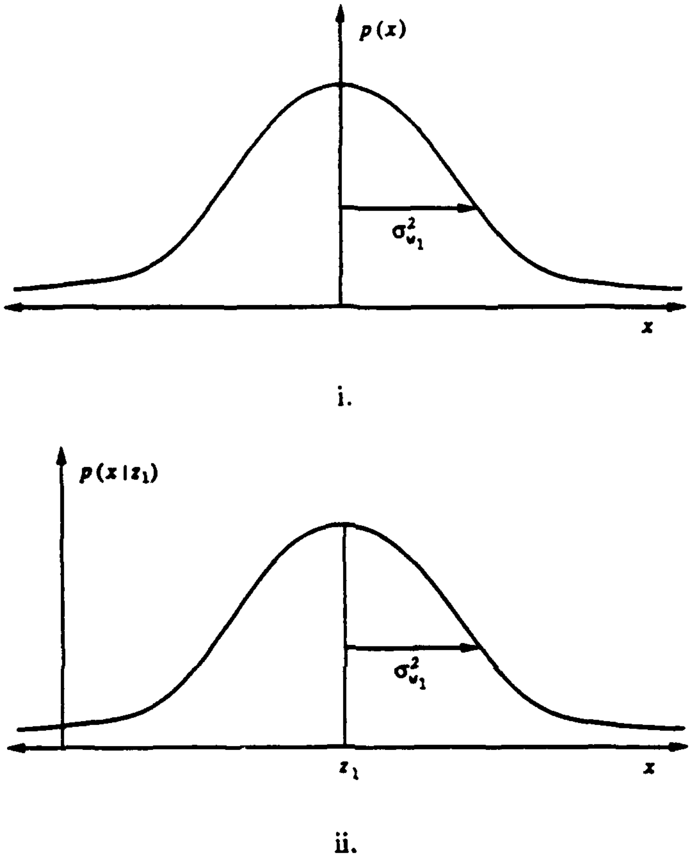


Figure 6.2: The densities for (i) the zero-mean Gaussian distribution $N(0, \sigma^2_{w_1})$ modeling the measurement noise for the first sensor, and (ii) the Gaussian distribution $N(z(1), \sigma^2_{w_1})$ modeling the measurement itself.

errors. Before we write down the equations for the Kalman filter, we consider some simple examples adapted from Maybeck [14] to illustrate the basic issues.

We return to our single-degree-of-freedom robot, moving back and forth on a horizontal track. Here we use the scalar x to represent the state of the system corresponding to the position of the robot on the track. Suppose that there are two sensors that allow the robot to obtain measurements its position. Each of the two sensors returns an estimate of the robot's location corrupted by Gaussian noise: $N(0, \sigma_{w_1}^2)$ in the case of the first sensor and $N(0, \sigma_{w_2}^2)$ in the case of the second. At time 1, the first sensor is deployed to obtain a measurement $z(1)$ of the robot's position. We model the measurement as a sum of the robot's actual position and the zero-mean Gaussian noise process shown in Figure 6.2.i. The conditional probability density for the actual position, x, given the measurement, $z(1)$, is shown in Figure 6.2.ii. The mean of the distribution is just $z(1)$ in this case, and the variance, $\sigma_{w_1}^2$, is rather large, indicating a sensor with significant potential for error.

Based on the density shown in Figure 6.2.ii, the best estimate of the robot's position is

$$\hat{x}(1) = z(1),$$

and the variance of the error in the estimate is

$$\sigma_x^2(1) = \sigma_{w_1}^2.$$

At time 2, following the first measurement and assuming that the robot has not moved, you obtain a second measurement, $z(2)$, from the second, and generally more reliable of the two sensors. The fact that this second sensor is generally more reliable is indicated by the density for the second measurement being more peaked (having a smaller variance) than the density for the first measurement as shown in Figure 6.3.i. In this case, the mean of the distribution is $z(2)$, and the variance is $\sigma_{w_2}^2$.

We can combine the two measurements to obtain a conditional density for the position of the robot given both measurements. The result is a Gaussian density, $N(\mu, \sigma^2)$, with mean, μ, given by

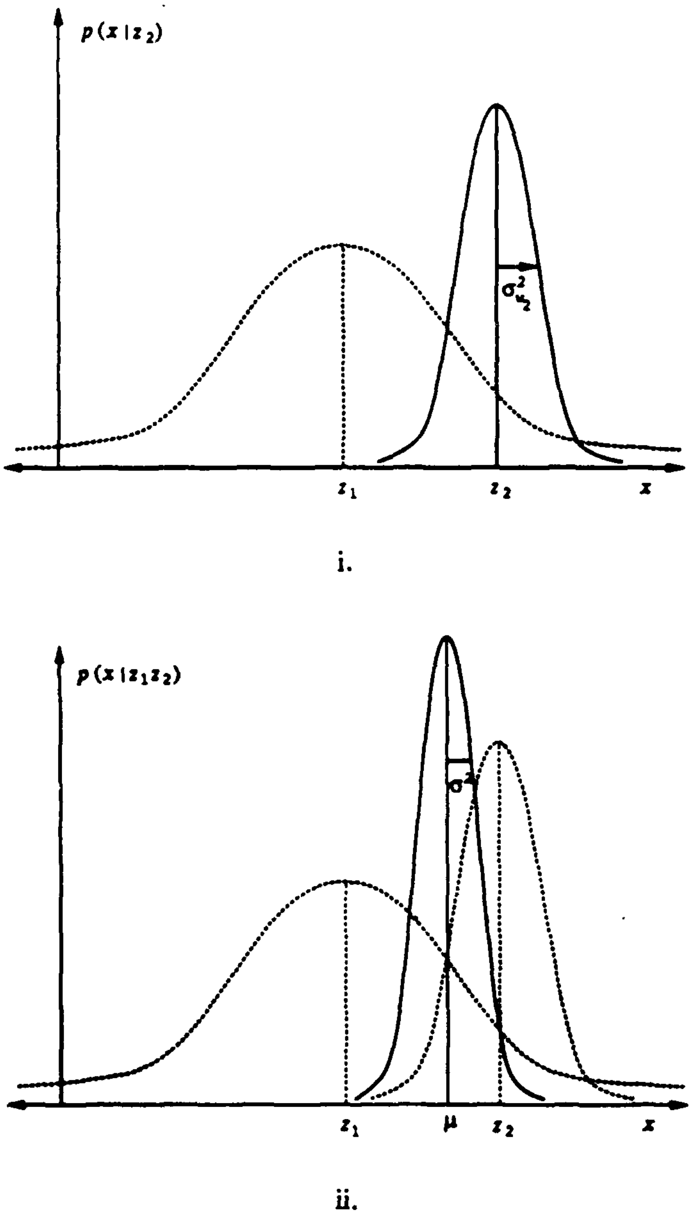
$$\mu = \left[\frac{\sigma_{w_2}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}\right] z(1) + \left[\frac{\sigma_{w_1}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}\right] z(2)$$

and variance, σ^2, given by

$$\sigma^2 = \frac{\sigma_{w_1}^2 \sigma_{w_2}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}.$$

Figure 6.3.ii depicts the resulting density superimposed over the densities for each of the individual measurements. Notice that $N(\mu, \sigma^2)$ is more peaked than either of the densities for the measurements taken separately. Given $N(\mu, \sigma^2)$, the best estimate for the robot's position at time 2 is

$$\hat{x}(2) = \mu,$$



i.

ii.

Figure 6.3: The densities for (i) the second measurement superimposed over the first, and (ii) the combined measurements superimposed over the first and second.

with an associated error variance

$$\sigma_x^2 = \sigma^2.$$

We will not provide a proof that this is the best estimate. We will, however, provide some intuitions as to why it is a plausible estimate.

The variances provide information to assist in establishing the relative weight to attach to the evidence from the previous measurement(s) and that from the latest measurement. If the two variances are equal, then the two measurements are equally reliable and we simply take their average. If, on the other hand, the variance for the previous measurement(s) is large and the variance for the latest measurement small, then we give more weight to the latest measurement. The variance will always decrease in the case of two or more measurements taken at the same time, reflecting the fact that additional (consistent) information should serve to sharpen the estimate. Casting the problem of state estimation in terms of optimization, the recursive update algorithm described in this section is optimal in the sense that it minimizes the variance.[4]

To adopt the form generally used in describing the Kalman filter, we rewrite the equation for $\hat{x}(2)$,

$$\begin{aligned}\hat{x}(2) &= \left[\frac{\sigma_{w_2}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}\right] z(1) + \left[\frac{\sigma_{w_1}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}\right] z(2) \\ &= z(1) + \left[\frac{\sigma_{w_2}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}\right] (z(2) - z(1))\end{aligned}$$

and, substituting $\hat{x}(1)$ for $z(1)$, we obtain

$$\hat{x}(2) = \hat{x}(1) + K(2)(z(2) - \hat{x}(1)),$$

[4] The variance is just the expectation of error. In the case of no prior expectations, we want to find the estimate, $\hat{x}$, minimizing the mean of the squared error,

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{x} - x_i)^2,$$

where the x_i are the measurements. We obtain this estimate by setting the derivative to zero,

$$\frac{d\left[\frac{1}{n}\sum_{i=1}^{n}(\hat{x} - x_i)^2\right]}{d\hat{x}} = 2\sum_{i=1}^{n}(\hat{x} - x_i) = 0,$$

and solving for $\hat{x}$. The estimate provided by the method described here is just the mean of the measurements, $\frac{1}{n}\sum_{i=1}^{n} x_i$, which is a solution to the above equation.

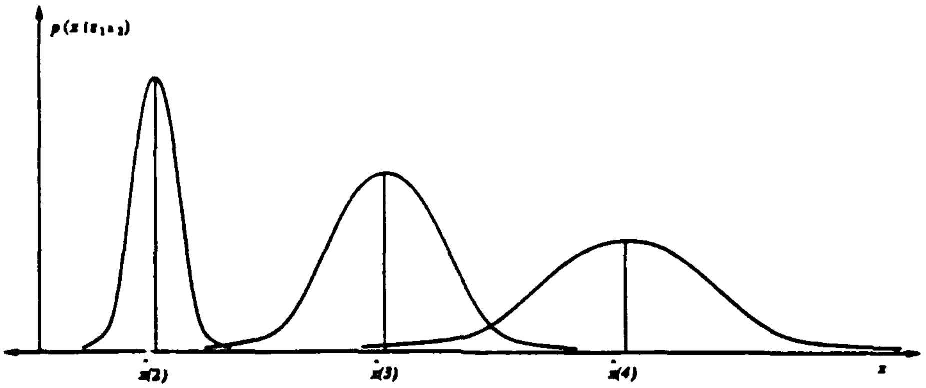


Figure 6.4: Evolving state estimates without additional measurements

where $K(2)$ is defined as

$$K(2) = \frac{\sigma_{w_2}^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2}.$$

Our objective is to provide an algorithm that computes an estimate of the evolving state of a dynamical system. We have not as yet made any real use of the equations describing the dynamical system. The method of combining measurements in the static case is generally referred to as *minimum mean-square estimation*, and is attributed to Carl Friedrich Gauss (1777–1855). The primary contribution of Kalman and the other researchers who developed and refined the Kalman filter is the recursive solution of minimum mean-square state estimation problems involving dynamical systems.

Given an estimate of the system state at time t, we wish to compute an estimate of system state at time $t+1$, which accounts for the most recent measurements *and* also for the system dynamics. Continuing with our example, we assume the following simple dynamics

$$x(k+1) = x(k) + u(k) + v(k),$$

where $u(t)$ is the distance moved, and $v(t)$ is a zero-mean, white, Gaussian noise process with variance, σ_v^2.

We denote the estimate of the system state at time 3 given only the measurements taken at time 2 or earlier as $\hat{x}(3|2)$ defined by

$$\hat{x}(3|2) = \hat{x}(2) + u(2),$$

with corresponding variance

$$\sigma_x^2(3|2) = \sigma_x^2(2) + \sigma_v^2.$$

If we made no additional measurements, the estimate of the system state would degrade over time, as shown in Figure 6.4. In general, however, we will make at least one measurement at every time step. To incorporate measurements taken at time 3, we employ the same basic equations used for combining $z(1)$ and $z(2)$.

Generalizing the previous examples, we present the Kalman filtering equations for the following one-dimensional dynamical system,

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) + v(k) \\ z(k) &= h(x(k)) + w(k), \end{aligned}$$

where x, u and z are scalar quantities, f and h are linear functions, and v and w are zero-mean, Gaussian noise processes with associated variance, σ_v^2 and σ_w^2 respectively. Since f and g are linear we can rewrite the above equations as

$$\begin{aligned} x(k+1) &= \overset{A}{C_1}x(k) + \overset{B}{C_2}u(k) + v(k) \\ z(k) &= \overset{C}{C_3}x(k) + w(k), \end{aligned}$$

where C_1, C_2, and C_3 are constants. We assume exactly one measurement taken at each time step.

Recall that the objective is to maintain an estimate of the state of the system at all times. The estimate of the system state at time k given all of the measurements up until time j is denoted $\hat{x}(k|j)$. Similarly, we denote the variance in the estimate at time k given all of the measurements up until time j as $\sigma_x^2(k|j)$. We write $\hat{x}(k|k)$ and $\sigma_x^2(k|k)$ simply as $\hat{x}(k)$ and $\sigma_x^2(k)$. At each time k, all of the past measurements are summarized by the estimate, $\hat{x}(k)$, and its associated variance, $\sigma_x^2(k)$.

There are three basic steps performed in updating the estimate of the system state to reflect the measurement made at $k+1$. These steps are referred to as the *prediction*, *observation*, and *estimation* steps. We consider each of them in turn.

In the prediction step, we compute what we expect to observe at $k+1$. This involves first computing an estimate of the state at $k+1$ given all the measurements at time k or earlier, defined by

$$\hat{x}(k+1|k) = \overset{A}{C_1}\hat{x}(k) + \overset{B}{C_2}u(k).$$

The variance associatec with this estimate is

$$\sigma_x^2(k+1|k) = C_1^{2}\sigma_x^2(k) + \sigma_v^2.$$

Notice that the control is not considered in computing the variance. The predicted measurement is then

$$\hat{z}(k+1|k) = C_2\hat{x}(k+1|k),$$

and the variance associated with the predicted measurement is

$$\sigma_z^2(k+1|k) = C_2^{2}\sigma_x^2(k+1|k) + \sigma_w^2.$$

In the observation step, we make the observation and then compare the resulting measurement with what we expected. The difference between the actual and predicted measurement,

$$\nu(k+1) = z(k+1) - \hat{z}(k+1|k),$$

is called the *innovation*.

In the third and final step, called the *estimation step*, we compute $\hat{x}(k+1)$ as

$$\hat{x}(k+1) = \hat{x}(k+1|k) + K(k+1)\nu(k+1),$$

and the associated variance as

$$\sigma_x^2(k+1) = \sigma_x^2(k+1|k) - (K(k+1))^2\sigma_z^2(k+1|k),$$

where $K(k+1)$ is called the *filter gain* and defined by

$$K(k+1) = \frac{\cancel{C_3}\sigma_x^2(k+1|k)}{\sigma_z^2(k+1|k) + \sigma_w^2(k+1|k)}$$

It should be noted that we have to invert the measurement function in order to compute the filter gain. In general, this inversion can be difficult if not impossible. However, for linear systems, inversion simply involves taking a reciprocal in the scalar case or inverting a matrix in the vector case.

A good way of convincing yourself that these equations make sense is to consider limiting cases. For instance, consider cases in which the there is no error in movement or measurement (*i.e.*, σ_v^2 and σ_w^2 are 0) or cases in which C_1, C_2, and C_3 are 1.

In the above, we made use of models for predicting not only the current and future states of the system, but also the current and future measurements made in observing the system. These models account for uncertainty in the underlying process by incorporating probabilistic noise models for disturbances in the dynamical system and errors in measurement. At each point in time, we compare what we expect to observe with what we actually observe in order to determine how much weight to attribute to each, based on the sort of errors we expect from the corresponding noise models.

Extending the above equations to handle finite vector spaces and multiple measurements is reasonably straightforward though notationally tedious, and we will not attempt it here. Instead of the mean and variance of the distribution of a single random variable, it is necessary to generalize to the mean and covariance of a multidimensional distribution of a vector of random variables.[5] Once you understand the equations for the single-dimensional case, it is relatively easy to understand the multidimensional case. It is quite another matter, however, to apply the equations to real problems which invariably deviate from the assumptions stated above. In the following, we consider some of the issues that arise in the application of the Kalman filter to robotics problems.

In many problems in robotics, linearity is hard to come by and one has to appeal to an extension of the Kalman filter designed to handle nonlinear state equations. For instance, in the case of even the simplest holonomic (turn-in-place) mobile robot, the state vector might consist of the robot's position along the x axis, its position along the y axis, and its orientation, θ, all specified with respect to some coordinate frame of reference in the

[5] For a vector, $\mathbf{x}$, of n random variables the n-dimensional normal (Gaussian) density is defined by

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|P|^{\frac{1}{2}}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu)'P^{-1}(\mathbf{x}-\mu)\right],$$

where μ and $P = E[(\mathbf{x}-\mu)(\mathbf{x}-\mu)']$ are the mean and covariance of the vector $\mathbf{x}$, and the prime (as in $(\mathbf{x}-\mu)'$) indicates vector (or matrix) transposition. The covariance of two random variables, x and y, indicates the degree to which x is related to y, and is defined by

$$E[(x - E(x))(y - E(y))] = E[xy] - E[x]E[y].$$

The covariance (matrix) of the n dimensional vector, $\mathbf{x}$, is the symmetric matrix whose ijth entry is the covariance of the ith and jth components of $\mathbf{x}$.

workspace:

$$\mathbf{x}(k) = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \end{bmatrix},$$

where we notate the state vector, $\mathbf{x}$, using a bold font to distinguish it from the state variable corresponding to position along the x axis. The input vector in this case is just

$$\mathbf{u} = \begin{bmatrix} D(k) \\ \Delta\theta(k) \end{bmatrix}$$

where $D(k)$ is the distance traveled in a single time step, and $\Delta\theta(k)$ is the rotation turned through in a single time step. We can write the state equation as

$$\begin{aligned} \mathbf{x}(k+1) &= f(\mathbf{x}(k), \mathbf{u}(k)) + \mathbf{v}(k) \\ &= \begin{bmatrix} x(k) + D(k)\cos\theta(k) \\ y(k) + D(k)\sin\theta(k) \\ \theta(k) + \Delta\theta(k) \end{bmatrix} + \mathbf{v}(k), \end{aligned}$$

which is clearly nonlinear.

The standard approach to dealing with such nonlinearities is to linearize the state equation by expanding the nonlinear function in Taylor series around the current estimate, $\hat{\mathbf{x}}$, with terms up to first or second order to obtain, respectively, the first- or second-order *extended Kalman filter.* In the case of the first-order extended Kalman filter for the nonlinear state equation above, we would have

$$\mathbf{x}(k+1) \doteq f(\hat{\mathbf{x}}(k), \mathbf{u}(k)) + f_{\mathbf{x}}(k)[\mathbf{x}(k) - \hat{\mathbf{x}}(k)] + \mathbf{v}(k),$$

where $f_{\mathbf{x}}(k)$ is the Jacobian matrix[6] of f defined by

$$f_{\mathbf{x}}(k) = \begin{bmatrix} 1 & 0 & -D(k)\sin\theta(k) \\ 0 & 1 & D(k)\cos\theta(k) \\ 0 & 0 & 1 \end{bmatrix}.$$

[6] The Jacobian is to vector-valued functions what the gradient is to scalar-valued functions. If f is a vector-valued function,

$$f(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_m(x_1, x_2, \ldots, x_n) \end{bmatrix},$$

Generally, the measurement functions are also nonlinear and require similar linearization. Having obtained the necessary linearizations, we then proceed as in the linear case, and hope that the resulting approximations will provide acceptable state estimates.

Modeling sensors so as to satisfy the Gaussian noise requirement is another problem frequently encountered in robotics applications. Most sensors cannot be modeled as simple functions of one or more of the state variables corrupted with Gaussian noise. Consider, for example, some of the problems that arise in modeling ultrasonic (sonar) sensors of the sort typically found on mobile robots.

A sonar sensor consists of an ultrasonic transducer, a receiver, and some signal-processing hardware. Information about the distance from the sensor to nearby surfaces is obtained by measuring the round-trip time of flight of an ultrasonic pulse that is emitted by the transducer, bounces off an object surface, and returns to the receiver.

If the transducer is pointed along a line perpendicular to a nearby planar surface, then the sensor can be modeled as the actual distance to the surface corrupted by zero-mean Gaussian noise However, if the transducer is not pointed perpendicular to the nearest object surface, then there is some chance that not enough of the energy from the ultrasonic pulse will be returned to the receiver to determine the true time of flight to the nearest surface. Instead, the pulse may be reflected, bouncing off possibly several objects before a signal with enough energy is detected by the receiver. In this case, the information returned by the sensor may deviate significantly from the distance to the nearest object. Figure 6.5 (from [13]) shows the range data obtained from a single sensor rotated 360°; the range data is superimposed over a line drawing of the room in which the sensor is located.

If you know that your sensor is pointing perpendicular to a planar surface, then you can use the Kalman filtering equations to obtain a good estimate of the distance separating the robot from the surface. The problem, of course, is that it is generally very difficult to know that you are

then its *Jacobian matrix* is defined by

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_1/\partial x_n \\ \vdots & \vdots & & \vdots \\ \partial f_m/\partial x_1 & \partial f_m/\partial x_2 & \cdots & \partial f_m/\partial x_n \end{bmatrix}.$$

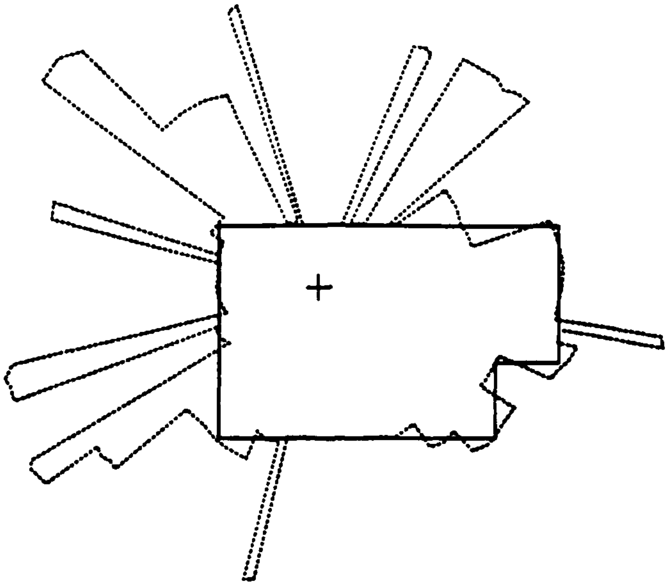


Figure 6.5: A 360° sonar scan of an indoor environment

pointing perpendicular to a planar surface.

If you have some *a priori* knowledge about the surfaces of the objects in the form of a map, then you can often make good guesses about what surfaces are out there and align your sensors so as to obtain reliable range data. In the following, we outline some basic steps in sonar guided navigation using an existing map and the Kalman filter.

1. Consult the map and extract some number of *beacons* corresponding to geometric features found in the map. This process of extracting beacons involves using the current estimate of the robot's state (position and orientation with respect to the frame of reference of the map). Useful geometric features are those whose sonar signature is distinctive. Flat walls (planar surfaces), round columns (cylindrical surfaces), and corners (intersection of planar surfaces) are examples of geometric features with distinctive sonar signatures. Having obtained a set of candidate beacons, we attempt to ascertain if they really stand in the expected relationship to the robot (and ultimately to one another).

2. For each candidate beacon, construct a model for the measurements that would be obtained from the sensor if the beacon was in the rel-

ative position and orientation predicted by the map. Note that the model may require that the sensor be aligned with the beacon in some particular configuration to avoid errors due to multiple reflections. We assume that there is a library of parameterized models, one for each type of geometric feature deemed useful. The model for a particular candidate beacon is obtained by instantiating one of the parameterized models using relative position and orientation information from the map. There would be a separate model for each beacon of the form

$$\mathbf{z}_i(k) = h_i(\mathbf{x}(k)) + \mathbf{w}_i(k),$$

where h_i is the nonlinear measurement function for the ith candidate beacon, and $\mathbf{w}_i$ models the measurement noise. Using the estimated state $\hat{\mathbf{x}}(k+1|k)$, we obtain a prediction for each observation

$$\hat{\mathbf{z}}_i(k+1|k) = h_i(\hat{\mathbf{x}}(k+1|k)).$$

3. We now make the next observations, using heuristic strategies where appropriate in an attempt to align the sensors according to the requirements of the corresponding model.[7] Given the actual and predicted observations, we compute the innovation

$$\nu_i(k+1) = \mathbf{z}_i(k+1) - \hat{\mathbf{z}}_i(k+1|k),$$

and the corresponding prediction variance which is obtained by linearizing the h_i. Up until this point, we have essentially followed the basic steps of the Kalman filter. However, in the next step, we deviate somewhat.

4. We have only hypothesized the existence of the candidate beacons, and we could easily turn out to be mistaken. Because of the possibility of making mistakes in identifying beacons, we cannot immediately use the innovations and their associated variances to obtain $\hat{\mathbf{x}}(k+1)$. It will not hurt if we are off a bit in our estimation of the geometric feature's relative location and orientation; the Kalman filtering

[7] Ideally the robot would be equipped with several independent rotating sensor arrays. Each array would consist of a pair of ultrasonic sensors mounted at some small distance apart on a rigid platform so that the two sensors are always pointing in the same direction. Each candidate beacon would be assigned an array and the beacon could then be aligned with the beacon surface(s) using a feedback controller that exploits the difference between values returned by the two sensors.

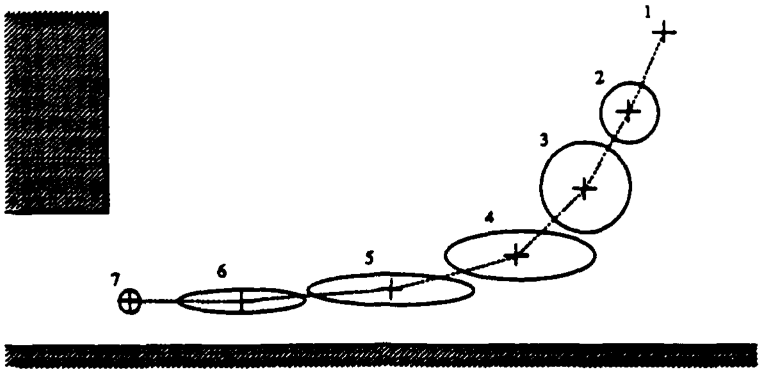


Figure 6.6: Localization using the extended Kalman filter

equations will weight the new measurements appropriately and, over time, the estimate should converge to the actual state. However, if the measurements are due not to the hypothesized beacon but rather to some other geometric feature, then incorporating those measurements into the state estimate using the Kalman filtering equations will lead to significant estimation errors. To avoid such errors, we subject the observations to the following test. We determine a range of possible values for each beacon such that, if the beacon is actually present, then the measurement will fall within that range with some reasonably high probability. We select only those measurements that fall within the range determined by the specified threshold probability.

5. Finally, we compute the latest estimate as

$$\hat{\mathbf{x}}(k+1) = \hat{\mathbf{x}}(k+1|k) + \sum_{i=1}^{m} K_i(k+1)\nu_i(k+1),$$

where K_i is the filter gain (matrix) for the ith measurement out of the m measurements obtained in the previous step.

The approach sketched above is conceptually quite simple but somewhat tricky to implement for a real robot. Determining an appropriate threshold probability requires a certain amount of experimentation. Achieving proper alignment is difficult in the case of highly specular (glossy) metal or painted surfaces. Unexpected objects, either moving or fixed but not accounted for in the map, can cause problems. If, however, there are plenty of potential beacons and there are enough sensors to track several of them at any o.e time, then quite robust performance can be achieved.

Figure 6.6 illustrates how the method described above would perform in a particular environment. The robot's location in the plane is represented at 7 discrete points in time. Initially, the robot knows its exact location with respect to the frame of reference of the global map. In the next two time steps, its estimated position becomes increasingly uncertain due to movement errors. This uncertainty is represented in Figure 6.6 in terms of ellipses corresponding to contours of constant probability of the error distribution. We assume that at time points 2 and 3 the robot is not tracking any beacons. At time point 4, the robot acquires a beacon corresponding to the wall shown at the bottom of Figure 6.6. This beacon allows the robot to decrease its uncertainty with respect to the y axis. The robot continues to track the wall beacon thereby obtaining an increasingly more accurate estimate for its position with respect to the y axis. At time point 6, the robot acquires the beacon corresponding to the corner at the left of Figure 6.6, obtaining more accurate estimates for its position with respect to the x axis.

This example illustrates a special case of a more general approach employing the Kalman filter as a basic subroutine. In the general approach, we assume that the world is in one of several states; it is our task to determine which is the actual state. For each of the possible states, we provide a dynamical model in terms of a linear system corrupted by Gaussian noise. For each model, we interpret the data as though produced by the model. We then choose the model whose predictions conform most closely to the data.

We had several motivations in presenting the material on state estimation and the Kalman filter. Mathematically, the Kalman filter is simple and elegant. Practically, the Kalman filter provides a powerful tool that can yield extremely precise and robust control systems. Approaches based on the Kalman filter are well suited for implementation on digital computers. They provide a disciplined approach to combining the data from any number of sources. Finally, the recursive update equations for the Kalman filter illustrate a cycle of activity involving prediction, observation, and estimation, that should play a part in any approach to dealing with uncertainty.

The state regulation problem for a linear dynamical system, quadratic performance index, linear control law, and Gaussian disturbance and measurement noise can be cast in terms of two separate problems. The problem of deterministic optimal control and the problem of stochastic optimal estimation. It has been shown that the two problems can be solved separately to yield an optimal solution to the combined control problem. While this

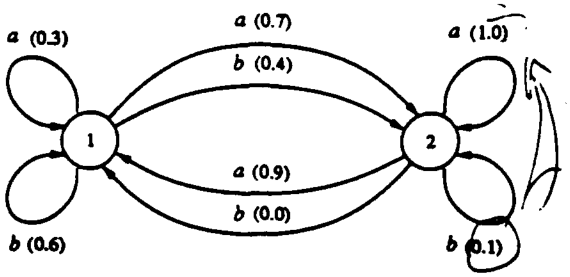


Figure 6.7: A stochastic process with two states

separation property does not hold for nonlinear systems, in many cases, engineers proceed as if it did, designing controllers and state estimators separately and then connecting them to obtain a complete control system. In estimation as elsewhere in control, the linear case serves as the basis for design. In Chapter 6 we consider problems in which observation and control interact strongly, requiring that the robot consider both state regulation and state reconstruction when choosing control actions.

6.3 Stochastic Dynamic Programming

In Chapter 3, we considered the problem of determining an optimal policy for multistage decision processes. In this section, we reconsider this problem in the context of *stochastic* processes. The material in this section is important in its own right, but it will also figure prominently in Chapters 6 and 8.

For our purposes, a finite-state, time-invariant, discrete-time *stochastic* process is a four tuple (T, X, U, P) consisting of the following.

- A set of time points $T = \mathbf{Z}$
- A finite set of states $X = \{x_1, x_2, \ldots, x_{|X|}\}$
- A finite set of inputs $U = \{u_1, u_2, \ldots, u_{|U|}\}$
- A set, $P = \{\rho_{ij}(u)\}$, of state-transition conditional probability distributions, one for each state/input pair, $\langle x_i, u\rangle$ where $x_i \in X$ and $u \in U$, such that for each $x_j \in X$ we have the distribution,

$$\rho_{ij}(u) = \Pr(x(t+1) = x_j | x(t) = x_i, u(t) = u),$$

independent of t, and subject to the standard requirements regarding probability distributions,

$$0 \leq \rho_{ij}(u) \leq 1, \quad \forall x_i, x_j \in X, u \in U,$$

and

$$\sum_{x_j \in X} \rho_{ij}(u) = 1, \quad \forall x_i \in X, u \in U.$$

We notate the state-transition distributions as $\rho_{ij}(u)$ so that in the sequel we can drop the explicit input argument by assuming an implicit control law or policy of the form,

$$\eta : X \rightarrow U,$$

so that

$$\rho_{ij} = \rho_{ij}(\eta(x_i)).$$

Figure 6.7 shows a simple stochastic process with two possible states, $X = \{1, 2\}$, and two possible inputs, $U = \{a, b\}$.

The stochastic processes we are considering here are guaranteed to transition to every state infinitely often no matter what initial state the process is started in. Such processes are said to be *completely ergodic*.

In addition to the requirements stated above, the stochastic processes that we will be concerned with have the following *Markov property*,

$$\Pr(x(t+1)|x(t), u(t)) = \Pr(x(t+1)|x(t), u(t), x(t-1), u(t-1), \ldots),$$

notation ✓

English paraphrase

indicating that the transition probabilities depends only on the last state and not on any prior history of the system.

Finally, we introduce a *reward* function,

$$\mathrm{R} : U \times X \rightarrow \mathbf{R},$$

such that $\mathrm{R}(u, x)$ corresponds to the (immediate) benefit derived from performing action u in state x. In Chapter 3, 4 we were concerned with n-stage decision problems and maximizing performance indices such as

$$\mathrm{V}(u(1), \ldots, u(n); x(1), \ldots, x(n)) = \sum_{i=1}^{n} \mathrm{R}(u(i), x(i)).$$

We were able to solve such problems using the following recurrence,

$$\begin{aligned} \mathrm{V}_n(x) &= \max_u [\mathrm{R}(u, x) + \mathrm{V}_{n-1}(f(x, u))], \quad n \geq 2 \\ \mathrm{V}_1(x) &= \max_u \mathrm{R}(u, x), \end{aligned}$$

where f is the deterministic state-transition function.

In the case of stochastic processes, there is generally some uncertainty in the outcome resulting from performing a given action in a particular state, and so we maximize *expected* value to account for this uncertainty. We can extend the recurrence for the deterministic case to handle stochastic processes by summing over the possible next states weighted by their probability of occurring. The extended recurrence is defined by

$$\begin{aligned} V_n(x_i) &= \max_u \sum_{x_j \in X} \rho_{ij}(u)[R(u, x_i) + V_{n-1}(x_j)], \quad n \geq 2 \\ V_1(x_i) &= \max_u \sum_{x_j \in X} \rho_{ij}(u)[R(u, x_j)]. \end{aligned}$$

The above recurrence represents the application of Bellman's principle of optimality, as discussed in Chapter 3, to Markov decision processes. The method of solving Markov decision processes by solving this recurrence is referred to as *value iteration* since the value functions are determined iteratively [9].

There are other variations on this basic recurrence relation. For instance, we could specify boundary conditions (*e.g.*, initial amount of fuel or other resource) by redefining V_1 to include some initial value. We could also define a set of admissible controls thereby restricting which actions are allowed under what circumstances. The primary limitation of value iteration concerns its ability to handle processes of indefinite duration. Under some circumstances the above recurrence can be shown to converge asymptotically, so that, in the limit as $n \rightarrow \infty$, an agent using the policy defined by

$$\eta(x_i) = \arg\max_u \sum_{x_j \in X} \rho_{ij}(u)[R(u, x_i) + V_{n-1}(x_j)],$$

will act so as to maximize its average expected return [3]. However, in certain cases, we can do much better, and, in the following, we consider a method due to Howard [9] for solving processes of indefinite duration.

If a completely ergodic stochastic process is allowed to transition indefinitely, the cumulative reward will increase without bound given a strictly positive reward function. A more appropriate performance index for processes of indefinite duration is the average reward per transition. We define the average reward per transition or *system gain* with respect to a given policy. In the following, we always assume a current policy of the form,

$$\eta : X \rightarrow U,$$

allowing us to make the following abbreviations,

$$\begin{aligned} \rho_{ij} &= \rho_{ij}(\eta(x_i)) \\ \mathrm{R}(x) &= \mathrm{R}(\eta(x), x) \end{aligned}$$

Using these abbreviations, we can rewrite the basic recurrence used in value iteration as follows.

$$\begin{aligned} \mathrm{V}_n(x_i) &= \sum_{x_j \in X} \rho_{ij}[\mathrm{R}(x_i) + \mathrm{V}_{n-1}(x_j)] \\ &= \sum_{x_j \in X} \rho_{ij}[\mathrm{R}(x_i)] + \sum_{x_j \in X} \rho_{ij}[\mathrm{V}_{n-1}(x_j)] \end{aligned}$$

We introduce new notation for the expected immediate (quick) returns corresponding to the first summation term in the above equation,

$$\mathrm{Q}(x_i) = \sum_{x_j \in X} \rho_{ij}[\mathrm{R}(x_i)],$$

allowing us to simplify the recurrence once more as

$$\mathrm{V}_n(x_i) = \mathrm{Q}(x_i) + \sum_{x_j \in X} \rho_{ij}[\mathrm{V}_{n-1}(x_j)].$$

Note that the quick returns can be computed directly from the reward function and the state-transition probabilities. To evaluate the quick return for an input other than that specified by the current policy, we simply add a control argument,

$$\mathrm{Q}(x_i, u) = \sum_{x_j \in X} \rho_{ij}(u)[\mathrm{R}(u, x_i)].$$

In considering processes with indefinite duration, we are interested in how often a given process will end up in a particular state. Let $\pi_i(n)$ indicate the probability that the system will be in state x_i after n transitions given that the initial state is known. Let π_i be the limit of $\pi_i(n)$ as $n \to \infty$. Clearly $\sum_{x_i \in X} \pi_i = 1$. For completely ergodic processes, the π_i are completely independent of the starting state and provide us with the frequency that the system will enter a given state given that it is allowed to run indefinitely. Using these limiting state transition probabilities, we can define the system gain (average reward per transition) with respect to a given policy as

$$\mathrm{G} = \sum_{x_i \in X} \pi_i[\mathrm{R}(x_i)].$$

As n gets large, the quantity, $\mathrm{V}_n(x)$, increases without bound, but the difference, $\mathrm{V}_n(x)-\mathrm{V}_{n-1}(x)$, is bounded. As a consequence, we can determine the equation of a line,

$$y(n) = gn + v_0,$$

bounding the values of $\mathrm{V}_n(x)$, where gn represents the steady-state component of the behavior as $n \rightarrow \infty$, and v_0 represents the transient component, depending only on the starting state. This bounding line is referred to as the *asymptote* of $\mathrm{V}_n(x)$. The slope, g, of the asymptote is just the system gain, G, and the y-intercept, v_0, we denote $\mathrm{V}(x)$ (no subscript) for starting state, x. For completely ergodic processes, the slope is independent of the starting state. As n gets large, we have the following approximation,

$$\mathrm{V}_n(x) = n\mathrm{G} + \mathrm{V}(x).$$

Substituting in our recurrence, we obtain

$$\begin{aligned} n\mathrm{G} + \mathrm{V}(x_i) &= \mathrm{Q}(x_i) + \sum_{x_j \in X} \rho_{ij}[(n-1)\mathrm{G} + \mathrm{V}(x_j)] \\ n\mathrm{G} + \mathrm{V}(x_i) &= \mathrm{Q}(x_i) + (n-1)\mathrm{G} \sum_{x_j \in X} \rho_{ij} + \sum_{x_j \in X} \rho_{ij}[\mathrm{V}(x_j)]. \end{aligned}$$

Noting that $\sum_{x_j \in X} \rho_{ij} = 1$, we finally obtain a set of equations of the form,

$$\mathrm{G} + \mathrm{V}(x_i) = \mathrm{Q}(x_i) + \sum_{x_j \in X} \rho_{ij}[\mathrm{V}(x_j)],$$

one for each $x_i \in X$. This constitutes a set of $|X|$ linear simultaneous equations in $|X|+1$ unknowns: the values of G and the $|X|$ $\mathrm{V}(x_i)$. In order to solve this system of equations, we can eliminate one unknown by setting one of the $\mathrm{V}(x_i)$ equal to zero. The values for the $\mathrm{V}(x_i)$ obtained from the solution to the set of simultaneous equation with, say, $\mathrm{V}(x_{|X|}) = 0$ will differ from those defined in

$$\mathrm{V}_n(x_i) = n\mathrm{G} + \mathrm{V}(x_i)$$

by a constant amount, but this difference is not significant for processes with a large number of transitions, and the values obtained for the $\mathrm{V}(x_i)$ will suffice for determining the relative merit of two policies, hence they are referred to as *relative values*.

We now have a method, referred to as *value determination*, for establishing the expected value of a given policy for a stochastic decision process of indefinite duration. We now need a method of choosing an optimal policy. In the following, we consider a method due to Howard [9] called *policy iteration* which allows us to generate an optimal policy by successive approximation. Policy iteration starts with an arbitrary policy, generates an improved (higher gain) policy on every iteration, and is guaranteed to terminate in a finite number of iterations with the optimal (highest possible attainable gain) policy. The policy iteration algorithm cycles between the value-determination procedure outlined above and a *policy-improvement* procedure that involves selecting an improved policy on the basis of the relative values for the current policy. As Howard [9] puts it, "the value-determination operation yields values as a function of policy, whereas the policy-improvement routine yields policy as a function of the values."

The policy iteration algorithm is defined as follows.

1. Let $k \leftarrow 0$.

2. Choose an arbitrary[8] policy, η_0, compute the corresponding values for the $Q(x_i)$, and then use the value determination method described above to compute the values for the $V(x_i)$.

3. For each state, x_i, find u_i maximizing

$$Q(x_i, u_i) + \sum_{x_j \in X} \rho_{ij}(u_i)[V(x_j)],$$

using the current value function. For each x_i, if u_i yields a better return based on the current value function, that is we have

$$\left(Q(x_i, u_i) + \sum_{x_j \in X} \rho_{ij}(u_i)[V(x_j)]\right) > \left(Q(x_i) + \sum_{x_j \in X} \rho_{ij}[V(x_j)]\right),$$

then $u_i' \leftarrow u_i$, otherwise $u_i' \leftarrow \eta_k(x_i)$.

[8] While the choice of initial policy does not affect whether or not the algorithm converges on the optimal policy, a good initial choice can often result in faster convergence. If there is no *a priori* reason for choosing any particular policy, Howard recommends choosing η_0 so that

$$\eta_0(x_i) = \max_u Q(x_i, u).$$

This is effectively the same as setting $V(x_i) = 0$ for all $x_i \in X$, and then running the policy improvement step in the algorithm.

4. Define a new policy such that

$$\eta_{k+1}(x_i) = u'_i.$$

5. If $\eta_k = \eta_{k+1}$, then exit returning η_k.

6. Using η_{k+1}, compute the values for the $Q(x_i)$, and then use these to compute $V(x_i)$ using value determination.

7. Let $k \leftarrow k + 1$.

8. Go to Step 3.

Step 6 and Step 2, both of which involve value determination, are the most expensive steps computationally. However, the solution of the set of simultaneous equations required for value determination can be easily handled by means of existing efficient linear programming algorithms. The limiting factor is the size of the state and input spaces.

To illustrate how policy iteration works, we consider a variation on a classic problem found in [9, 3]. The classic formulation involves a taxicab driver searching for fares; we have changed the problem slightly to reflect our interest in mobile robots. Our treatment here follows that of [9].

Consider the problem faced by a robot courier assigned the task of delivering files, office supplies, and other assorted small items in a three-story office building. The robot is rewarded for making its deliveries and the rewards differ depending on where the robot is and how far it is required to travel.

For the most part, the robot just waits around for the next delivery job, but it has a few options that can influence how quickly the next job arrives and how much of a reward it is likely to obtain in carrying out this job. Each floor of the building is dedicated to a different department of a company, and each floor has its own separate reception area and copy room. The offices on the first and third floors are equipped with computer workstations linked by local area networks, and the robot can plug into the network on a given floor using a receptacle located near the elevator. Using their personal workstations, office workers can issue requests to the robot through the network.

Let $X = \{1,2,3\}$, corresponding to the first, second, and third floors of the office building, and $U = \{c,r,n\}$, corresponding to the three options open to the robot, wait in the copyroom, wait in the reception area, and

i	u	$\rho_{ij}(u)$			$R_{ij}(u)$			$Q_i(u)$
		$j =$ 1	2	3	$j =$ 1	2	3	
1	c	1/2	1/4	1/4	10	4	8	8.00
	r	1/16	3/4	3/16	8	2	4	2.75
	n	1/4	1/8	5/8	4	6	4	4.25
2	c	1/2	0	1/2	14	0	18	16.00
	r	1/16	7/8	1/16	8	16	8	15.00
3	c	1/4	1/4	1/2	10	2	8	7.00
	r	1/8	3/4	1/8	6	4	2	4.00
	n	3/4	1/16	3/16	4	0	8	4.50

Table 6.1: Specification for the robot-courier problem

plug into the local area network, where the last option is only available in States 1 and 3. Since the reward depends not only on the action taken and the initial state, but also upon the final state, we modify the reward function to take a third argument, $R : U \times X \times X \rightarrow \mathbf{R}$, so that $R_{ij}(u)$ corresponds to the (immediate) benefit derived from performing action u in state x_i and ending up in state x_j. We also modify the definition of the immediate (quick) reward function to reflect the dependence on the final state,

$$Q_i(u) = \sum_{x_j \in X} \rho_{ij}(u)[R_{ij}(u)].$$

The complete specification for the robot courier problem is shown in Table 6.1 where the transition probabilities and rewards are shown in matrix form.

We begin by assuming that the expected values for all states are zero,

$$V(1) = V(2) = V(3) = 0,$$

so that the initial policy will depend only upon immediate rewards. Looking at the last column in Table 6.1, it should be clear that the robot should wait in the copyroom no matter what floor it finds itself on, and so we define the initial policy, η_0, as

$$\eta_0(1) = \eta_0(2) = \eta_0(3) = c.$$

The state transition probabilities and reward values for this policy are given by the following matrices,

$$[\rho_{ij}] = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{bmatrix} \qquad [\mathrm{R}_{ij}] = \begin{bmatrix} 8 \\ 16 \\ 7 \end{bmatrix}.$$

From the general equations used in value determination,

$$\mathrm{G} + \mathrm{V}(x_i) = \mathrm{Q}_i + \sum_{x_j \in X} \rho_{ij}[\mathrm{V}(x_j)],$$

we construct the particular equations for the current policy,

$$\begin{aligned} \mathrm{G} + \mathrm{V}(1) &= 8 + \frac{1}{2}\mathrm{V}(1) + \frac{1}{4}\mathrm{V}(2) + \frac{1}{4}\mathrm{V}(3) \\ \mathrm{G} + \mathrm{V}(2) &= 8 + \frac{1}{2}\mathrm{V}(1) + 0\mathrm{V}(2) + \frac{1}{2}\mathrm{V}(3) \\ \mathrm{G} + \mathrm{V}(3) &= 8 + \frac{1}{4}\mathrm{V}(1) + \frac{1}{4}\mathrm{V}(2) + \frac{1}{2}\mathrm{V}(3) \end{aligned}$$

Setting V(3) equal to zero and solving, we obtain

$$\begin{aligned} \mathrm{V}(1) &= 1.33 \\ \mathrm{V}(2) &= 7.47 \\ \mathrm{V}(3) &= 0 \\ \mathrm{G} &= 9.2 \end{aligned}$$

Table 6.2 shows the results of the calculations made in the process of improving upon the initial policy, η_0. For each state, x_i, we choose the option, u, that maximizes the quantity,

$$\mathrm{Q}_i(u) + \sum_{x_j \in X} \rho_{ij}(u)[\mathrm{V}(x_j)],$$

and select the improved policy, η_1, defined by

$$\eta_1(1) = c, \quad \eta_1(2) = r, \quad \eta_1(3) = r,$$

indicating that the robot should wait in the reception area on the second and third floor, but wait in the copyroom on the first.

i	u	$Q_i(u) + \sum_{x_j \in X} \rho_{ij}(u)[V(x_j)]$
1	c	10.53
	r	8.43
	n	5.52
2	c	16.67
	r	21.62
3	c	9.20
	r	9.77
	n	5.97

Table 6.2: First round of policy improvement for the robot courier

If we perform another cycle of value determination and policy improvement, we arrive at the policy, η_2 defined by

$$\eta_2(1) = r, \quad \eta_2(2) = r, \quad \eta_2(3) = r,$$

indicating the robot should wait by in the reception area no matter what floor it is located on. If we perform yet another cycle we obtain, η_3, defined by

$$\eta_3(1) = r, \quad \eta_3(2) = r, \quad \eta_3(3) = r.$$

Noticing that $\eta_2 = \eta_3$, we now have an optimal policy,

$$\eta(1) = r, \quad \eta(2) = r, \quad \eta(3) = r,$$

for the robot courier problem, reinforcing the belief held by many office workers that the reception area is one of the busiest areas in an office and one to be avoided if you wish to avoid work.

As might be expected, policy iteration is sensitive to a variety of changes in the initial conditions. For instance, if you reverse the transition probabilities, $\rho_{2,2}(r)$ and $\rho_{2,3}(r)$, you obtain a different optimal policy,

$$\eta(1) = c, \quad \eta(2) = c, \quad \eta(3) = r.$$

In addition, the number of iterations (most importantly, the number of times we have to perform value determination) depends critically on the choice of an initial policy. If, for example, we start with the initial policy,

$$\eta_0(1) = n \;\; \eta_0(2) = r \;\; \eta_0(3) = n,$$

policy iteration takes only two iterations instead of the three required for $\eta_0(1) = \eta_0(2) = \eta_0(3) = c$. In many cases, the choice of an initial policy that is close to optimal can improve the performance of policy iteration dramatically.

In some cases, it is unrealistic to count consequences in the distant future on an equal basis with more immediate consequences. For instance, we may mistrust our model for making accurate long term predictions, or future rewards may actually lose value due to some inflationary process. Most biological organisms tend to discount longer term rewards and focus on more immediate rewards. We can model this outlook on rewards by adding a discounting factor to our value function.

$$V_n(x_i) = Q(x_i) + \lambda \sum_{x_j \in X} \rho_{ij}[V_{n-1}(x_j)],$$

where $0 \leq \lambda < 1$ is the *discount rate.* In the case of discounting, the notion of gain (average reward per transition) no longer makes sense, as the optimal policy is simply the one that maximizes expected value in all possible states.

Value determination is actually simpler for stochastic processes with discounting, as we no longer have to account for the system gain. Eliminating the system gain and appealing once more to the asymptotic limit of V_n, namely V, we obtain a set of equations of the form,

$$V(x_i) = Q(x_i) + \lambda \sum_{x_j \in X} \rho_{ij}[V(x_j)],$$

one for each $x_i \in X$. This constitutes a set of $|X|$ linear simultaneous equations in $|X|$ unknowns (the $V(x_i)$) that can be easily solved for the unknowns. Policy iteration works in the case of discounting exactly as before with the substitution of the simplified value determination procedure.

If we add discounting to the robot-courier problem, we get a different policy depending upon the value of λ. For $0 \leq \lambda < 0.13$, we get the policy,

$$\eta(1) = c, \quad \eta(2) = c, \quad \eta(3) = c,$$

for $0.13 \leq \lambda < 0.53$, we get

$$\eta(1) = c, \quad \eta(2) = r, \quad \eta(3) = c,$$

for $0.53 \leq \lambda < 0.77$, we get

$$\eta(1) = c, \quad \eta(2) = r, \quad \eta(3) = r,$$

and, finally, for $0.77 \leq \lambda < 1.0$, we get

$$\eta(1) = r, \quad \eta(2) = r, \quad \eta(3) = r,$$

As one might guess, the closer λ is to 1, the more iterations of value determination and policy improvement will be required to obtain the optimal policy.

In Chapter 8, we consider a form of learning that is closely related to the approach used here to compute an optimal policy for stochastic decision processes with discounting. We will employ the same basic form of successive policy improvement. The main departure from the techniques of this section is that value determination will be done without the aid of ~~model~~ knowledge of the transition probabilities and reward function. Value determination will occur over time as the agent interacts with its environment obtaining rewards and punishments intermittently and occasionally inappropriately. This sort of *reinforcement learning* provides a good model of learning in biological organisms and also appears to be a good model for many automated planning and control applications.

6.4 Fuzzy Set Theory and Fuzzy Control

Uncertainty arises in many different forms. Probability theory provides a basis for reasoning about uncertainty due to randomness, but there are other forms of uncertainty that cannot be easily captured using the tools of probability theory. In this section, we consider some alternative tools provided by *fuzzy set theory* and *fuzzy control*.

Fuzzy set theory provides a mathematical basis for capturing knowledge in a form close to that used in everyday communication. Using fuzzy set theory, we can assign meaning to terms associated with sets for which there are no clearly defined boundaries separating elements from non-elements, terms like large, small, close, far, hot, cold, short, and tall.

The standard interpretations of probabilities in terms of frequencies or likelihoods make it difficult to model linguistic phenomena characterized by words like "heavy" or "tall." The word "tall" denotes a fuzzy set not because there is randomness in the process of measurement, but because there is general dispute and uncertainty about whether a borderline case belongs to the set or not.

Our interest here stems from the considerable success that fuzzy set theory and its counterpart, fuzzy control, have had in practical applications. Fuzzy control systems have been used in video cameras, automobiles, and high-speed public transportation systems, just to name a few of the more successful applications. Fuzzy control and fuzzy decision-support systems provide a focus on knowledge acquisition and representation similar to that found in the work on so-called *expert* rule-based systems. We mention fuzzy methods in this chapter because they have shown themselves to provide a viable alternative to other more traditional approaches to dealing with uncertainty in control, and because they share with other rule-based approaches to reasoning an emphasis on symbolic representations.

We begin with a brief introduction to fuzzy set theory [17]. Let X denote the *universe* set of elements, and x an instance of this set. A *fuzzy set A* in X is characterized by a membership or *characteristic* function from X to the real interval [0, 1],

$$\mathcal{I}_A : X \rightarrow [0,1].$$

The value of $\mathcal{I}_A$ at x indicates the "degree" to which x is considered to be a member of A. In standard set theory, $\mathcal{I}_A$ is either 0 or 1. In the sort of sets that fuzzy set theory is primarily concerned with, such binary distinctions are often difficult to make. For instance, let X be the set of all people, and A be the set of "tall" people. Suppose you consider people over seven feet to be tall, under six feet not to be tall, and between six and seven feet to be to some degree (between zero and one) tall. In this case, you might characterize the set of tall people using the following function,

$$\mathcal{I}_A(x) = \begin{cases} 1 & \text{if } 7 \geq h(x) \\ h(x) - 6 & \text{if } 6 \geq h(x) < 7 \\ 0 & \text{otherwise} \end{cases},$$

where $h(x)$ denotes the height of x.

We now provide fuzzy versions of some common set-theoretic notions. The fuzzy complement, $\bar{A}$, of the set, A, is defined by the function,

$$\mathcal{I}_{\bar{A}}(x) = 1 - \mathcal{I}_A(x).$$

The fuzzy union, $A \cup B$, of two fuzzy sets, A and B, is defined

$$\mathcal{I}_{A \cup B}(x) = \max(\mathcal{I}_A(x), \mathcal{I}_B(x)),$$

and the fuzzy intersection, $A \cap B$, is defined

$$\mathcal{I}_{A \cap B}(x) = \min(\mathcal{I}_A(x), \mathcal{I}_B(x)).$$

Note that, in the case of boolean-valued characteristic functions, these definitions coincide with the standard set-theoretic definitions of complement, union, and intersection.

For building rule-based control systems, we are not so much interested in a generalization of set theory as we are in a generalization of predicate logic. The standard (Tarskian) semantics for predicate logic is based on standard set theory; predicates denote sets, the (truth-functional) interpretation of atomic sentences is defined in terms of membership, and the meaning of the connectives, $\neg$, $\vee$ and $\wedge$, defined, respectively, in terms of complementation, union, and intersection. In a similar manner, one can provide semantics for *fuzzy logic* using fuzzy set theory. Since our objectives in this section are modest, we only introduce those concepts that are necessary for our discussion, and refer the reader to a more detailed treatment in [10].

The syntax for the propositional case is as follows. Let $\mathcal{A}$ be a set of fuzzy propositional variables. We define the set of well-formed formulae (wffs) inductively as consisting of any propositional variable, the negation of any wff (written $\neg\varphi$ where φ is a wff), the conjunction of any two wffs (written $(\varphi_1 \wedge \varphi_2)$ where φ_1 and φ_2 are wffs), or the disjunction of any two wffs (written $(\varphi_1 \vee \varphi_2)$ where φ_1 and φ_2 are wffs).

Next, we provide the semantics for the propositional case. An *interpretation*, M, is a function from propositional variables to the real interval $[0,1]$. An interpretation, M, is said to be an *α-model* for a wff, φ, (written $M \models_\alpha \varphi$) under the following conditions.

- $M \models_\alpha A$ iff $M(A) = \alpha$, where $A \in \mathcal{A}$
- $M \models_\alpha \neg\varphi$ iff $M \models_{1-\alpha} \varphi$
- $M \models_\alpha \varphi_1 \wedge \varphi_2$ iff $\alpha = \min(\alpha_1, \alpha_2)$, where $M \models_{\alpha_1} \varphi_1$, and $M \models_{\alpha_2} \varphi_2$,

In analogy to two-valued propositional logic, a wff, φ, is said to be *α-satisfiable* if it has an α-model, and is said to be *α-valid* (written $\models_\alpha \varphi$) if all models are α-models. We can also define an analog of semantic entailment. A wff, φ_1, is said to *α-entail* another wff, φ_2, (written $\varphi_1 \models_\alpha \varphi_2$) if for any model, M, $M \models_{\alpha_1} \varphi_1$ implies that $\alpha_2/\alpha_1 \geq \alpha$, where $M \models_{\alpha_2} \varphi_2$.

For a particular control problem, we would construct a set of fuzzy propositional variables as follows. Let F be the set of fuzzy sets, and X the universe set (generally the state space of a dynamical system) for the problem at hand. For each $A \in F$ and $x \in X$, we define a propositional variable of the form, $A(x)$, as shorthand for $x \in A$.

Assigning $A(x)$ a real number is like assigning a proposition a truth value; such assignments restrict the interpretations we are willing to consider and therefore restrict what formulae are valid. In a two-valued propositional logic, if you are told that P must be true in all interpretations, then, subject to that restriction, Q is true in all models for $\neg P \vee Q$. Similarly, in fuzzy logic, if you are told that $A(x)$ must be assigned 0.7 in all interpretations, then $B(x)$ must be assigned 0.4 in all models for which $\neg A(x) \vee B(x)$ is assigned 0.4. Note that, in the case of boolean-valued characteristic functions, the above fuzzy semantics reduces to standard truth-functional semantics.

For the cases we consider in the sequel, we are interested in the unique model, M, such that, for all $A \in F$ and $x \in X$, $M(A(x)) = \mathcal{I}_A(x)$. Illustrating the connection to the fuzzy set-theoretic concepts introduced earlier, note that M satisfies the following conditions

$$
\begin{aligned}
M &\models_{\mathcal{I}_{A}(x)} \neg A(x) \\
M &\models_{\mathcal{I}_{A\cap B}(x)} A(x) \wedge B(x) \ , \\
M &\models_{\mathcal{I}_{A\cup B}(x)} A(x) \vee B(x)
\end{aligned}
$$

for all $A, B \in F$ and $x \in X$.

The primitive notions presented above provide us with all the logical machinery we require for building simple fuzzy control systems. We could use fuzzy logic directly to obtain assignments to fuzzy propositional variables in an analog of the way in which boolean logic is used in some control systems. Instead, we consider how fuzzy logic formulae are used to construct *fuzzy algorithms* [18]. For our purposes, a fuzzy control system consists of a set of statements (or rules) of the form,

$$\text{If } A_1 \wedge A_2 \wedge \cdots \wedge A_n, \text{ then } C.$$

where the A_i are the antecedent conditions and C is the consequent action. Generally, the antecedents correspond to fuzzy propositions involving the system state variables, and the consequent corresponds to a fuzzy assignment statement involving the system input variables.

For instance, suppose you are trying to control a robot to move parallel to the planar surface of a wall in the direction right facing the wall and maintaining a distance of about one meter from the wall. You would need fuzzy sets characterizing the distance separating the robot from the wall, and the angle of the robot with respect to the surface of the wall. The distance to the wall might be captured using six fuzzy sets, corresponding to being next to the wall, *VERY_NEAR*, some distance but close, *NEAR*, somewhat further but still relatively close, *SOMEWHAT_NEAR*, even further,

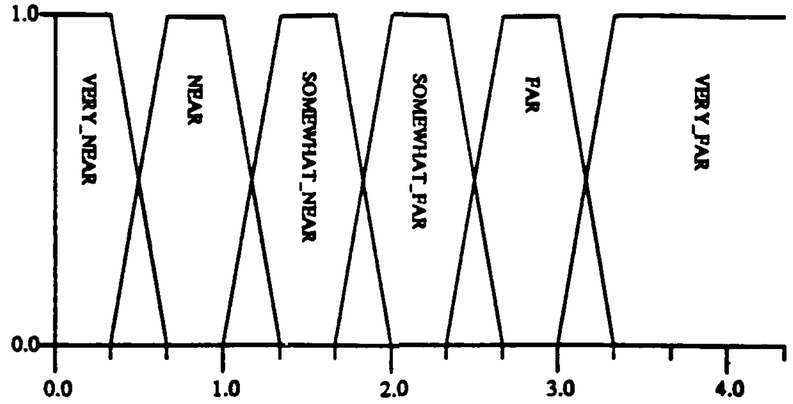


Figure 6.8: Fuzzy membership functions for the wall-following problem

SOMEWHAT_FAR, further still, *FAR*, and very far *VERY_FAR*. Possible characteristic functions for these six fuzzy sets are shown in Figure 6.8.

To control the robot, you might specify that, if the robot is within a meter or so of the wall and moving nearly perpendicular but slightly toward the wall's surface, then steer a little further to the right. Such a specification would be represented by the rule,

$$\begin{aligned}\text{R1:}\quad &\text{If } \mathit{NEAR}(x) \wedge \mathit{SOMEWHAT\_TOWARD}(x)\\ &\text{then } u \leftarrow u + \mathit{SOMEWHAT\_RIGHT},\end{aligned}$$

where *NEAR*, *SOMEWHAT_TOWARD*, and *SOMEWHAT_RIGHT* correspond to fuzzy sets, x is the system state indicating the position and orientation of the robot with respect to the wall, and u is the system input indicating the steering angle.

Fuzzy logic indicates how to interpret the antecedent of R1. For instance, given that

$$\begin{aligned}\mathit{NEAR}(x) &= 1.0\\ \mathit{SOMEWHAT\_TOWARD}(x) &= 0.9,\end{aligned}$$

we have

$$\mathit{NEAR}(x) \wedge \mathit{SOMEWHAT\_TOWARD}(x) = 0.9.$$

However, the statements in a fuzzy algorithm are not formulae in a fuzzy logic. What we require is a procedural interpretation. In particular, we have to determine the *result* of executing R1?

If *SOMEWHAT_RIGHT* were a constant, say 5°, then the result of executing R1, might be that the value of u is increased by 5 over what it was formerly, where the general rule might be, if the value of the antecedent is greater than 0.75, then treat the consequent as a statement in a conventional programming language and execute it accordingly.

In the case of *SOMEWHAT_RIGHT* being a fuzzy set, we will want to consider a different evaluation strategy. Suppose that we define the fuzzy set, *SOMEWHAT_RIGHT*, as follows,

$$\mathcal{I}_{SOMEWHAT\_RIGHT}(x) = \begin{cases} 0.2 & \text{if } x = 1^\circ \\ 0.4 & \text{if } x = 2^\circ \\ 0.6 & \text{if } x = 3^\circ \\ 0.8 & \text{if } x = 4^\circ \\ 1.0 & \text{if } x = 5^\circ \\ 0.8 & \text{if } x = 6^\circ \\ 0.6 & \text{if } x = 7^\circ \\ 0.4 & \text{if } x = 8^\circ \\ 0.2 & \text{if } x = 9^\circ \\ 0.0 & \text{otherwise} \end{cases}$$

Then we might define the result of executing R1 as another fuzzy set, *RESULT_R1*, defined by weighting the fuzzy set, *SOMEWHAT_RIGHT*, using the value assigned to the antecedent condition.

$$\mathcal{I}_{RESULT\_R1}(x) = \begin{cases} 0.2 * 0.9 & \text{if } x = u + 1^\circ \\ 0.4 * 0.9 & \text{if } x = u + 2^\circ \\ 0.6 * 0.9 & \text{if } x = u + 3^\circ \\ 0.8 * 0.9 & \text{if } x = u + 4^\circ \\ 1.0 * 0.9 & \text{if } x = u + 5^\circ \\ 0.8 * 0.9 & \text{if } x = u + 6^\circ \\ 0.6 * 0.9 & \text{if } x = u + 7^\circ \\ 0.4 * 0.9 & \text{if } x = u + 8^\circ \\ 0.2 * 0.9 & \text{if } x = u + 9^\circ \\ 0.0 & \text{otherwise} \end{cases}$$

We still need a unique result, and one obvious possibility is to choose the result with the highest rating, breaking ties randomly if necessary.

The method of using thresholds to determine whether or not to execute the consequent of fuzzy rules is inadequate in the case in which there are several rules all attempting to perform conflicting actions, say setting a control variable to different values, and all having antecedent co¬ditions that pass the threshold. For instance, in addition to R1, we might have the following rule,

$$\begin{array}{ll} \text{R2:} & \text{If } NEAR(x) \wedge SOMEWHAT\_AWAY(x) \\ & \text{then } u \leftarrow u + SOMEWHAT\_LEFT. \end{array}$$

As an alternative to thresholds, we could define a corresponding fuzzy result, *RESULT_R2*, for R1, and set u according to the following,

$$u \leftarrow \arg\max_{x}(\mathcal{I}_{RESULT\_R1}(x), \mathcal{I}_{RESULT\_R2}(x)).$$

We can generalize on the above method for any number of rules. In practice, the set of rules is represented using an n-dimensional table, with one dimension for each state variable and some number of fuzzy sets to cover the domain of each such variable. At each point in time, all of the rules are evaluated to determine their corresponding fuzzy results, and the maximal control action taken.

More About the semantic entailment & fuzzy control rules V

There are many different schemes for executing fuzzy algorithms. There are methods that combine the results from several rules, using a variety of weighting schemes. There are fuzzy algorithmic versions of integer programming, dynamic programming, database query processing, as well as a host of specialized techniques for financial decision making, natural-language processing, circuit layout, and speech recognition, just to name a few. Our purpose here is not to survey fuzzy methods, but simply to make the reader aware of a large and active area of control, and provide a somewhat different perspective on uncertainty than that offered by the probabilists.

6.5 Further Reading

For a more thorough treatment of state estimation techniques in general and the Kalman filter in particular, the reader is encouraged to read Bar-Shalom and Fortmann [1], Brammer and Siffling [4], Gelb [6], or Maybeck [14]. It is also well worth returning to some of the original papers on the theory and application of the Kalman filter. A number of the original papers appear in a collection by Sorenson [16] which is particularly interesting for the broad range of applications considered.

For approaches to geometrical reasoning under uncertainty involving static estimation and using minimum mean-square parameter estimation techniques, see the work of Durrant-Whyte [5] and Smith and Cheeseman [15]. Hager [7] presents a game-theoretic analysis of the errors that arise in applying minimum mean-square estimation methods and develops alternative techniques for stochastic geometrical reasoning that allow more flexibility in modeling uncertainty.

Leonard and Durrant-Whyte [13] describe techniques to obtain estimates of the distance separating a mobile robot from nearby walls, corners, and other environmental features that exhibit well-behaved sonar signatures. These estimates are then used to update the robot's position with respect to a global map. The discussion in Section 6.2 is based on their work.

While there are any number of more recent books on dynamic programming and stochastic decisi·›n processes, the texts by Bellman [2] and Bellman and Dreyfus [3] are well worth reading. The method of policy iteration discussed in this chapter is due to Howard [9], and his book is an excellent source of examples as well as proofs of correctness for the basic method and a number of interesting variations. Among the variations, Howard discusses nonergodic (multichain) and continuous-time processes. For an introduction to finite Markov processes. the texts by Kemeny and Snell [11] and Hoel, Port, and Stone [8] are recommended.

The original paper by Zadeh [17] is still an excellent introduction to fuzzy set theory. In a later paper, Zadeh [18] considers the use of fuzzy set theory for reasoning about complex systems and decision processes. In this same paper, Zadeh elaborates on the notion of a fuzzy algorithm, providing a number of interesting examples. The text by Kaufman [10] covers some of the mathematics of fuzzy logic and fuzzy set theory.

Bibliography

[1] Bar-Shalom, Yaakov and Fortmann, Thomas E., *Tracking and Data Association*, (Academic Press, New York, 1988).

[2] Bellman, Richard, *Dynamic Programming*, (Princeton University Press, 1957).

[3] Bellman, Richard and Dreyfus, Stuart, *Applied Dynamic Programming*, (Princeton University Press, Princeton, New Jersey, 1962).

[4] Brammer, Karl and Siffling, Gerhard, *Kalman-Bucy Filters*, (Artech House, Norwood, Massachusetts, 1989).

[5] Durrant-Whyte, Hugh F., *Integration, Coordination and Control of Multi-Sensor Robot Systems*, (Kluwer Academic Publishers, Boston, Massachusetts, 1988).

[6] Gelb, A., (Ed.), *Applied Optimal Estimation*, (MIT Press, Cambridge, Massachusetts, 1974).

[7] Hager, Gregory D., *Task-Directed Sensor Fusion and Planning: A Computational Approach*, (Kluwer Academic Publishers, Boston, Massachusetts, 1990).

[8] Hoel, Paul G., Port, Sidney C., and Stone, Charles J., *Introduction to Stochastic Processes*, (Houghton Mifflin, Boston, Massachusetts, 1971).

[9] Howard, Ronald A., *Dynamic Programming and Markov Processes*, (MIT Press, Cambridge, Massachusetts, 1960).

[10] Kaufmann, Arnold, *Introduction to the Theory of Fuzzy Subsets*, (Academic Press, New York, 1975).

[11] Kemeny, J. G. and Snell, J. L., *Finite Markov Chains*, (D. Van Nostrand, New York, 1960).

[12] Koditschek, D., Robot Control Systems, Shapiro, Stuart, (Ed.), *Encyclopedia of Artificial Intelligence*, (John Wiley and Sons, New York, 1987), 902–923.

[13] Leonard, John J. and Durrant-Whyte, Hugh F., *Active Sensor Control for Mobile Robotics*, Technical Report OUEL-1756/89, Oxford University Robotics Research Group, 1989.

[14] Maybeck, Peter S., *The Kalman Filter—An Introduction to Potential Users*, Technical Report TM-72-3, Air Force Flight Dynamics Laboratory, Wright Patterson AFB, Ohio, 1972.

[15] Smith, Randall and Cheeseman, Peter, On the Representation and Estimation of Spatial Uncertainty, *The International Journal of Robotics Research*, 5 (1986) 56–68.

[16] Soronson, Harold W., (Ed.), *Kalman Filtering: Theory and Application*, (IEEE Press, New York, 1985).

[17] Zadeh, Lofti A., Fuzzy Sets, *Information and Control*, 8 (1965) 338–353.

[18] Zadeh, Lofti A., Outline of a New Approach to the Analysis of Complex Systems and Decision Processes, *IEEE Transactions on Systems, Man and Cybernetics*, 3 (1973) 28–44.

Chapter 7

Planning Under Uncertainty

This chapter is still very much in flux. It currently consists of early drafts of a couple of introductory sections along with some example sections drawn verbatim from conference and journal papers. No further apologies will be made for its state of disarray.

The approaches to planning that we considered in earlier chapters involve generating possible states of affairs from some initial information and a model. In this and the next two chapters, we focus on problems in which the present and future states of affairs are not completely determined by the model and the information at hand. We have already seen some problems of this sort. In the case in which a robot is uncertain of the outcome of an action, but the outcome will be apparent once the action is completed, we suggested that the robot construct a conditional plan indicating what subsequent course of action to take for each possible outcome. In this chapter, we consider cases in which the agent has somewhat more information about the possible outcomes before the action is completed, and somewhat less information about the actual outcome after the action is completed.

7.1 Decision Theory

Let Ω be a set of possible states. Suppose that we have some means of assigning numerical values to possible states:

$$V : \Omega \to \mathbf{R}.$$

[0] ©1990 Thomas Dean. All rights reserved.

This function is generally referred to as a *value* or *utility* function. In some cases, depending on our measure of value, it may be more convenient to think of value in terms of its inverse, *cost*. In the case of value or utility, we generally seek to increase it; in the case of cost, we generally seek to decrease it.

If you could choose some $\omega \in \Omega$, you would want to choose ω such that $V(\omega)$ is maximum:

$$\arg\max_{\omega\in\Omega} V(\omega).$$

Unfortunately, we cannot simply select at will from Ω. We assume, however, that we can select our actions from a set of actions, $\mathcal{A}$. Let $[\alpha|\omega]$ denote the state resulting from executing action α in state ω. If the state is unimportant or clear from context, we simply write $[\alpha]$.

Suppose that each action $\alpha \in \mathcal{A}$ has a unique outcome $[\alpha] \in \Omega$. Then we could simply choose the action whose outcome is most desirable:

$$\arg\max_{\alpha\in\mathcal{A}} V([\alpha]).$$

Of course, an action seldom, if ever, completely determines a unique state. To represent an agent's uncertainty about the consequences of its actions, we assume that the state resulting from a given action is governed by a random process. In this case, we let $[\alpha]$ denote a random variable with probability space Ω, and assume that we have conditional probability distributions of the form:

$$\Pr([\alpha]|\mathcal{E}),$$

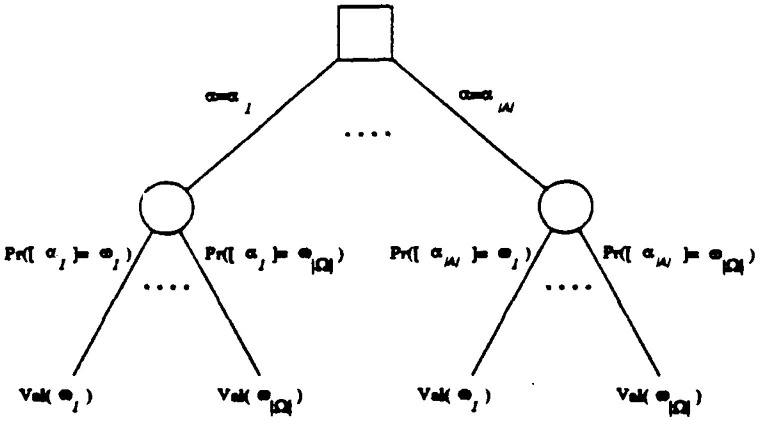
for all $\alpha \in \mathcal{A}$, where $\mathcal{E}$ represents the agent's background knowledge. Now, $V([\alpha])$ is a real-valued function of a random variable, and its expectation is defined to be:

$$E(V([\alpha])|\mathcal{E}) = \sum_{\omega\in\Omega} V(\omega)\Pr([\alpha] = \omega|\mathcal{E}). \qquad (7.1)$$

The agent will want to choose the action with the highest expected value:

$$\arg\max_{\alpha\in\mathcal{A}} E(V([\alpha])|\mathcal{E}). \qquad (7.2)$$

One assumption underlying the decision strategy captured in Equations 7.1 and 7.2 is that the agent is often going to find itself in the situation of having to choose what action to take. Hence, the agent wants to choose actions so that its long-term payoff, as predicted by the value function and



$\alpha=\alpha_1$
$\alpha=\alpha_{|A|}$
....
$\Pr([\alpha_1] \models \omega_1)$
....
$\Pr([\alpha_1] \models \omega_{|\Omega|})$
$\Pr([\alpha_{|A|}] \models \omega_1)$
....
$\Pr([\alpha_{|A|}] \models \omega_{|\Omega|})$
$Val(\omega_1)$
$Val(\omega_{|\Omega|})$
$Val(\omega_1)$
$Val(\omega_{|\Omega|})$

Figure 7.1: Simple decision tree

its expectations concerning outcomes is maximized. A decision that maximizes expected value is called an *optimal* decision.

A significant portion of the next two chapters will involve variations on this basic idea of choosing actions on the basis of expectations about their outcomes. so it is important that you understand it. You can picture the decision process embodied in Equations 7.1 and 7.2 as a *decision tree*. in which the root node corresponds to a choice by the agent of what action to take. and the children of the root node correspond to a choice by nature of what state should result from the agent's action. Figure 7.1 shows a simple decision tree in which the agent's choices are represented by boxes called *decision nodes*. and nature's choices are represented by circles called *chance nodes*. The terminal nodes in the decision tree are labeled with the values assigned to the outcomes. The edges leading out of chance nodes are labeled with the probabilities of the outcomes. The edges leading out of decision nodes are labeled with the agent's choices.

In general. decision trees can be of any depth. not just depth 2 as in the decision tree shown in Figure 7.1. Often decision trees are arranged with levels alternating between decision and chance nodes. but this is not required. There is no requirement that decision trees be symmetrical though they often appear so in textbooks. Indeed. we will often sacrifice symmetry to reduce the size of the decision tree and the computational effort required to evaluate the optimal decision.

An example should help to make the approach to decision making de-

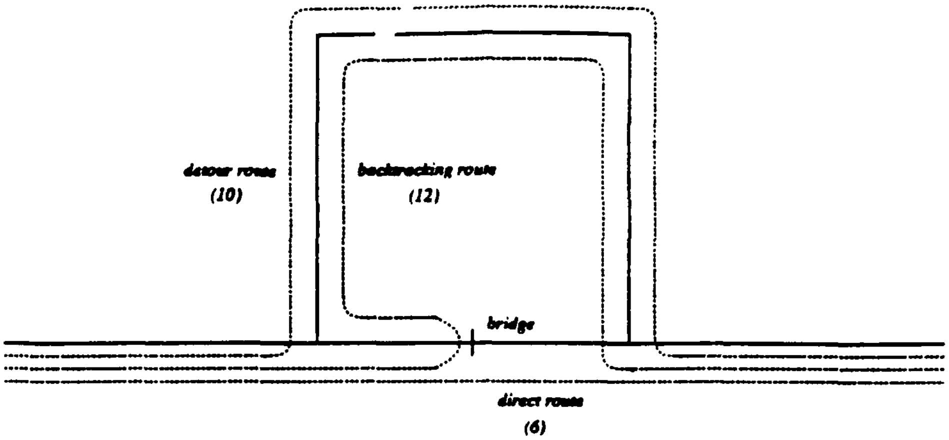


Figure 7.2: Alternative routes to the beach

scribed here more concrete. Suppose that you live in the city and are taking your summer vacation at a beach some distance from the city. Suppose further that there are two routes to the beach: a direct route that takes six hours and roundabout route that takes ten hours. We will call these the *direct* and *detour* routes. The direct route requires that you cross a bridge which, as luck would have it, is undergoing major repairs this summer. There is a 50% chance that the bridge will be closed at the time you wish to cross it. If you attempt the direct route and find the bridge closed, you will have to backtrack to the detour route, and your total transit time will be twelve hours.

Your decision involves choosing whether to try the direct or detour route first. Figure 7.2 shows the three possible outcomes of your decision. If you choose the detour route, the trip will take ten hours. If you choose the direct route, the trip will take either six hours or twelve hours depending on whether or not the bridge is closed. We need to assign a value or cost to each of the possible outcomes, and, in this case, a natural measure of cost is time spent in transit.

Figure 7.3 provides a graphical representation of the decision problem for choosing which route to take. Note that the terminals of the subtree emanating from the end of the detour branch have the same cost. The probabilities on the edges of this subtree govern whether or not the bridge is closed, but this factor has no impact on the outcome if we take the detour. Such uninteresting subtrees are generally eliminated and replaced with the

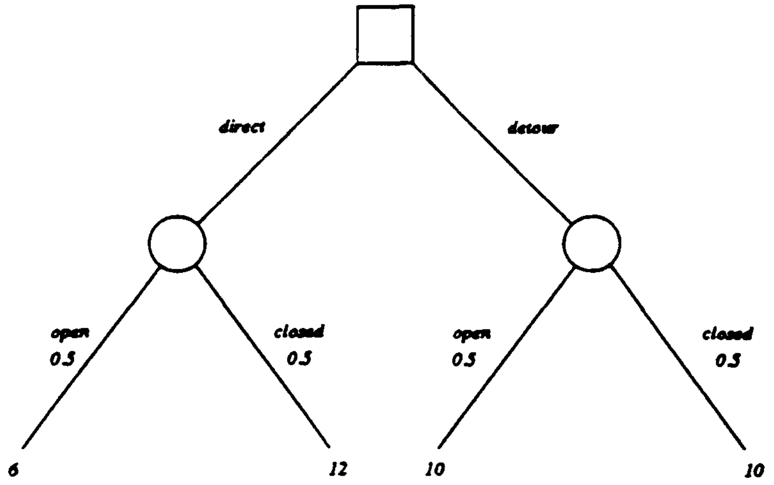


Figure 7.3: Decision tree for the vacation trip problem

value of the appropriate outcome.

Given a decision tree such as that depicted in Figure 7.3. we can calculate the optimal decision and its expected cost using the following simple procedure. Initially, all of the nodes in the tree except terminal nodes have null labels. Terminal nodes are labeled with the cost of outcomes.

1. For each chance node with a null labei all of whose children have non null labels. label it with the expected cost for the node calculated as the sum over all children of the product of the probability of the child (as indicated on the edge from the chance node to the child) and the child's label.

2. For each decision node with a null label all of whose children have non null labels, label it with the minimum cost of the labels of its children, and strike from consideration all edges except that one leading to the child with minimum cost.

3. If there are any nodes with null labels. go to Step 1, otherwise find a path from the root to a terminal node consisting of action edges that have not been stricken from consideration. The sequence of actions along this path indicates the optimal decision and its expected cost is the label of the root.

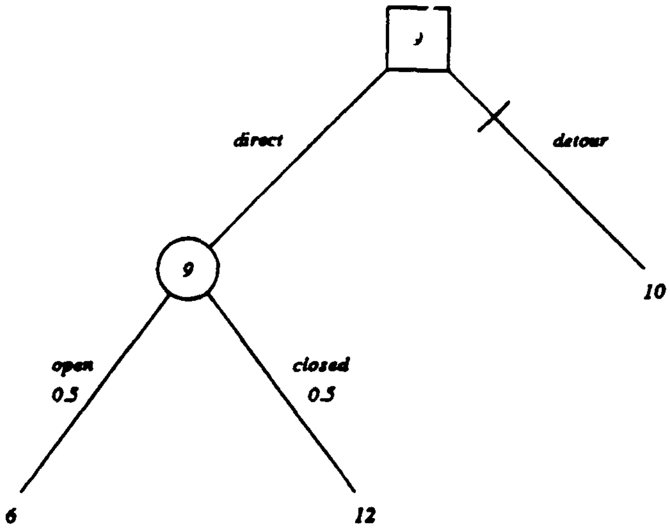


Figure 7.4: Evaluated decision tree for the vacation trip problem

If we are concerned with value instead of cost, substitute cost everywhere for value, and maximum and maximize everywhere for minimum and minimize.

Figure 7.4 shows the labeled and marked decision tree for the vacation trip problem obtained using the above procedure. The optimal decision is to try the direct route first, and the expected transit time in this case is 9 hours.

We can extend the above analysis to handle sequences of actions of length n. Let $\bar{\alpha} = \alpha_1, \alpha_2, \ldots, \alpha_n$, where $\bar{\alpha} \in \mathcal{A} \times \mathcal{A} \times \ldots \mathcal{A}$. The result of executing the sequence of actions $\alpha_1, \alpha_2, \ldots, \alpha_n$ in ω is

$$[\alpha_n|[\alpha_{n-1}|[\ldots[\alpha_1|\omega]\ldots]]],$$

abbreviated $[\alpha_1, \alpha_2, \ldots, \alpha_n|\omega]$. We denote the kth action in the sequence $\bar{\alpha} = \alpha_1, \alpha_2, \ldots, \alpha_k, \ldots, \alpha_n$ as $\bar{\alpha}_k$. The corresponding decision tree is shown in Figure 7.5. There are two things to note about the tree shown in Figure 7.5. First, the tree is likely to be quite large, $O(|\mathcal{A}|^n|\Omega|)$ nodes. Second, the tree is not very interesting in terms of capturing the structure of the decision problem. We might as well just use the simple two-level decision tree shown in Figure 7.1, and let the choice of what action to take range over the complex actions in $\mathcal{A} \times \mathcal{A} \times \ldots \mathcal{A}$.

There are cases, however, in which actions can alter an agent's decision making capability by providing additional information. For instance, if

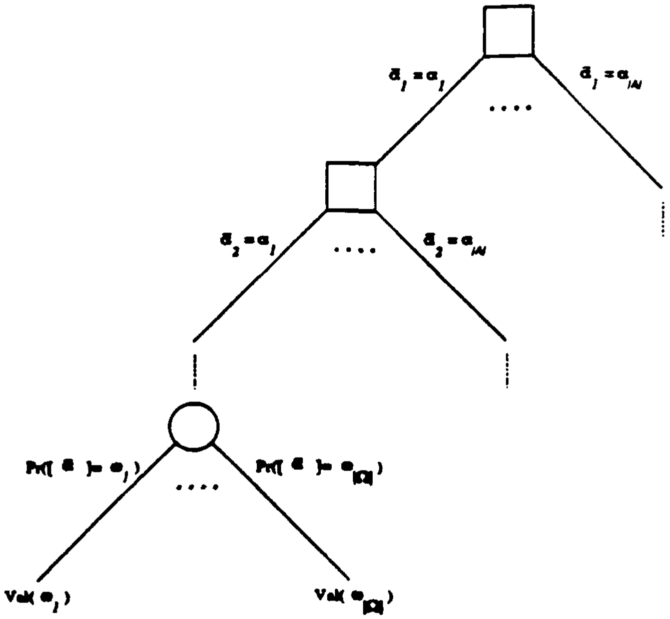

$\bar{\alpha}_1 = \alpha_1$
$\bar{\alpha}_1 = \alpha_{|A|}$
....
$\bar{\alpha}_2 = \alpha_1$
$\bar{\alpha}_2 = \alpha_{|A|}$
....
$Pr([\bar{\omega}] = \omega_1)$
$Pr([\bar{\omega}] = \omega_{|\Omega|})$
....
$Val(\omega_1)$
$Val(\omega_{|\Omega|})$

Figure 7.5: Sequential Decision Tree

you are interested in buying a used car, hiring a mechanic to check the car's condition before making a purchase will probably reduce the possibility that you end up buying a car with high repair costs. By representing the consequences of such information-gathering actions explicitly in our graphical representations for decision problems, we can gain some additional insight into the structure of such problems.

In our vacation trip example, suppose there is a state police station located near the highway prior to the point at which we have to decide between the direct and detour routes. We will assume that the state police can provide us with information about the current status of the bridge. Suppose that stopping at the police station requires getting off the highway and traveling to a nearby town, and that the total time spent in acquiring the information about the bridge is estimated to be 30 minutes.

Now we have an additional decision to make besides simply whether to take the direct or detour route. You can think of the trip to the police station as particular type of test with two possible findings: the state police believe that the bridge is open or they believe that it is closed. The findings may not provide conclusive evidence with regard to the primary question we are interested in. namely whether or not the bridge is closed. but let us suppose in this case that the beliefs of the state police are veridical.

We represent possible findings of our test as a chance node in the decision tree. The probabilities correspond to our priors regarding the status of the bridge. since at this time we have no better information. Under each of the two possible findings, we attach the tree shown in Figure 7.3 with one change: the probabilities for the chance nodes corresponding to whether or not the bridge is closed are now conditioned on the findings. Given our assumption that the police know the true status of the bridge, the probability the bridge is closed given the police say it is closed is 1, the probability the bridge is closed given the police say it is open is 0, and so on.

Figure 7.6 shows the decision tree for the vacation trip problem with the decision node corresponding to driving to the police station or not. The two options are labeled *check* and *not check*. We also label test options with their associated costs. Information costs. Every time that you get operator assistance in dialing a long-distance number or consult an accountant about your income tax you are paying for information. In the vacation trip problem, the cost of the information regarding the status of the bridge is in terms of increased driving time: 1/2 hour for the *check* option and no increase in time for the *not check* option. In computing the optimal decision for a decision problem with decision nodes corresponding to tests. we calculate the maximum values for the labels of such nodes accounting for these costs.

In the case of decision problems with actions to acquire information. the optimal decision is a conditional plan specifying what to do at each point in time given the information available at the time. This conditional plan is called the *optimal policy* in the decision sciences. The optimal policy for the decision tree shown in Figure 7.6 is to check with the state police, and then take the direct route if the police say the bridge is open and the detour otherwise. For this policy, the expected transit time is 8 and 1/2 hours.

It is often useful to be able to assess the value of information so as to make reasonable decisions regarding whether or not to pay for it. We can quantify the value of information in decision-theoretic terms.

In the vacation trip example, we were able to compute the expected value of making the trip by selecting actions that minimize expected travel time based on the information at hand. Let

$$E(T|\mathcal{E})$$

be the expected travel time, T. for the optimal course of action based on

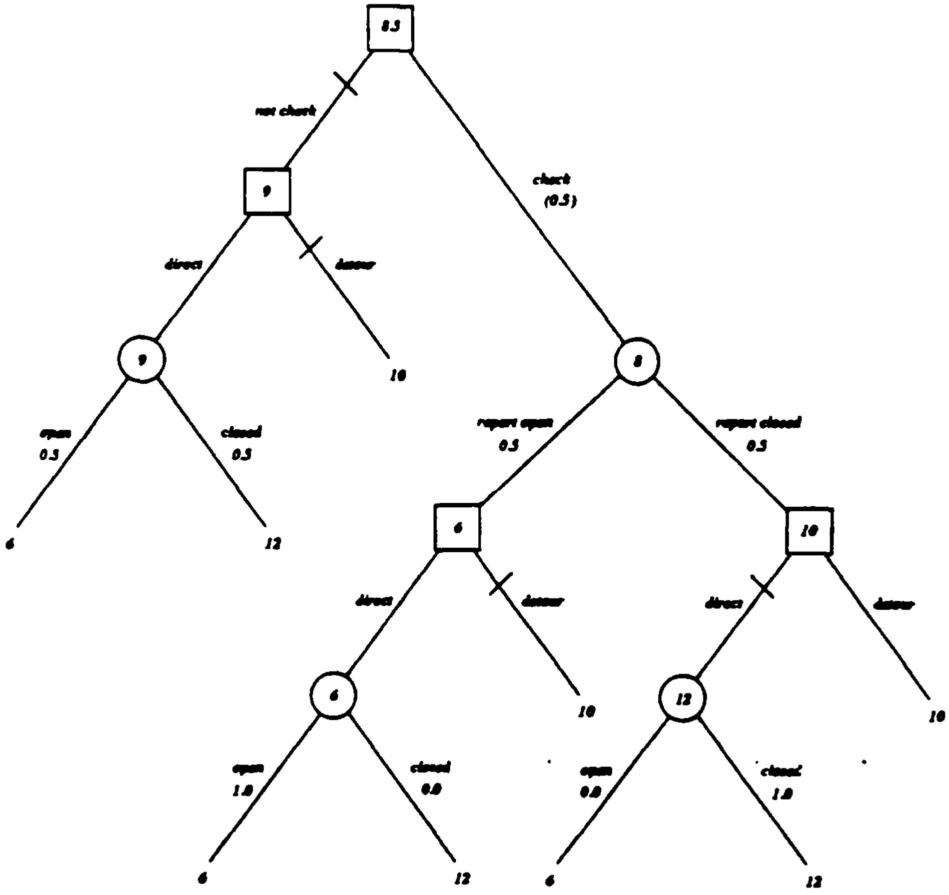


Figure 7.6: Reasoning about information gathering actions

the background information, $\mathcal{E}$. In reasoning about whether or not to stop at the state police station, we computed the expected travel time given the additional information obtained from the police:

$$E(T|I_S, \mathcal{E}).$$

where I_S represents the event of obtaining information from the police regarding the status, S, of the bridge, either *open* or *closed*. The expected value of the information obtained from stopping at the police station is

$$E(V(I_S)|\mathcal{E}) = E(T|I_S, \mathcal{E}) - E(T|\mathcal{E}),$$

where

$$E(T|I_S, \mathcal{E}) =$$
$$E(T|S = closed, \mathcal{E})\,\Pr(S = closed|\mathcal{E}) + E(T|S = open, \mathcal{E})\,\Pr(S = open|\mathcal{E}).$$

In the example, $E(V(I_S)|\mathcal{E}) = 1.0$, implying that we should be willing to spend up to one hour to obtain the information regarding the status of the bridge.

More generally, let $E(V([])|\mathcal{E})$ be the expected value of carrying out your present policy. Suppose that, prior to carrying out your present policy, someone offers to sell you information pertaining to some variable, X, used in calculating $E(V([]|\mathcal{E})$. To be more specific, suppose that the informant is clairvoyant and knows the actual value of X. Let I_X correspond to the event of obtaining the information regarding X.

The expected value of obtaining this information is given by

$$E(V(I_X)|\mathcal{E}) = E(V([])|I_X, \mathcal{E}) - E(V([])|\mathcal{E}). \qquad (7.3)$$

To compute $E(Val([])|I_X, \mathcal{E})$, we evaluate the expectation given knowledge about X for each possible value of X provided by the informant, summing over these expectations weighted by our prior on X

$$E(V([])|I_X, \mathcal{E}) = \sum_{x \in \Omega_X} E(V([])|X = x, \mathcal{E})\,\Pr(X = x|\mathcal{E}). \qquad (7.4)$$

It is important to note, as did Howard in the 1966 paper [19] in which he introduced Equations 7.3 and 7.4, that we use the prior distribution $\Pr(X|\mathcal{E})$ for X because, until the informant provides the information about X, our knowledge of X is based entirely on our background knowledge $\mathcal{E}$.

A good deal of the discussion in this and the next chapter will concern reasoning about the value of information and using the results of this reasoning to direct action. Before we can progress much further, we need to provide some additional machinery for probabilistic reasoning. In the next section, we consider a particular framework for modeling the world in the presence of uncertainty. We show how this framework can be extended to handle decision making, and then we demonstrate the power of the extended framework using applications involving sensing and mobile robotics.

Have to establish a generic name for what have been called Bayes nets, Bayesian networks, belief networks, probabilistic networks, influence diagrams and who knows what else.

7.2 Probabilistic Networks

A probabilistic network is a directed acyclic graph $\mathcal{G} = (V, E)$, where V is a finite set of *vertices*, and E, the set of *edges*, is a subset of $V \times V$, the set of ordered pairs of distinct vertices. Before we discuss how to use these probabilistic networks to build decision models, we introduce and define some standard graph theoretic terms.

If $(v_1, v_2) \in \mathcal{G}$, then v_1 is said to be a *parent* of v_2, and v_2 a *child* of v_1. The set of all parents of v is denoted Pa(v) and the set of all children Ch(v).

A path of length n from v_0 to v_n is a sequence $v_0, v_1, \ldots, v_n$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, n$. If there is a path from v_1 to v_2, then v_1 is said to be an *ancestor* of v_2. The set of all ancestors of v is denoted An(v). A subset $S \subseteq V$ is said to *separate* $V' \subseteq V$ from $V'' \subseteq V$ if every path from a vertex in V' to a vertex in V'' intersects S.

We can obtain an undirected graph from $\mathcal{G}$ by ignoring the ordering on the pairs of vertices in E. The graph so obtained is called the *undirected graph corresponding to* $\mathcal{G}$. If $V' \subseteq V$, then V' induces a subgraph $\mathcal{G}_{V'} = (V', E_{V'})$ where $E_{V'}$ is that subset of E restricted to $V' \times V'$. A graph (V, E) is *complete* if for all $v_1, v_2 \in V$ either $(v_1, v_2) \in E$ or $(v_2, v_1) \in E$. If $V' \subseteq V$ induces a complete subgraph, then V' is said to be complete. A complete subset that is maximal with respect to set inclusion is called a *clique*.

For the directed acyclic graph (V, E), we define its *moral graph* as the undirected graph with the same vertex set in which v_1 is adjacent to v_2 just in case either $(v_1, v_2) \in E$, $(v_2, v_1) \in E$, or there exists $v_3 \in V$ such that both $(v_1, v_3) \in E$ and $(v_2, v_3) \in E$.

The vertices in V correspond to random variables and are called *chance* *nodes* as in decision trees. The edges in E define the causal and informational dependencies between the random variables. In the models described here, chance nodes are discrete-valued variables that encode states of knowledge about the world. We use upper-case italic letters (*e.g.*, X) to represent random variables, and lower-case italic letters (*e.g.*, x) to represent their possible values. Let Ω_X denote the set of possible values (*state space*) of the chance node X. In order to *quantify* a probabilistic network, we have to specify a probability distribution for each node. If the chance node has no parents, then this is its unconditional (*marginal*) probability distribution, $\Pr(X)$; otherwise, it is a conditional probability distribution dependent on the states of the parents, $\Pr(X|\text{Pa}(X))$.

If $V = \{X_1, X_2, \ldots, X_n\}$, we can write down the joint distribution using the chain rule as follows:

$$\Pr(X_1, X_2, \ldots, X_n) =$$
$$\Pr(X_n|X_{n-1}, \ldots, X_1)\Pr(X_{n-1}|X_{n-2}, \ldots, X_1)\cdots\Pr(X_2|X_1)\Pr(X_1).$$

There are certain independence assumptions implicit in the structure of probabilistic networks that enable us to simplify this expression somewhat. A complete characterization of the conditional independencies embodied in the structure of a given probabilistic network can be given in graph theoretic terms. For a given $\mathcal{G} = (V, E)$ and subsets $V', V'', S \subseteq V$, V' is conditionally independent of V'' given S if S separates V' from V'' in the moral graph for $\mathcal{G}$. From this characterization, it follows that a chance node is conditionally independent of its ancestors given its parents:

$$\Pr(X|\text{An}(X)) = \Pr(X|\text{Pa}(X)).$$

If the indices, $1, \ldots, n$, of the variables, $X_1, X_2, \ldots, X_n$, are consistent with the partial ordering in $\mathcal{G}$ (*i.e.*, $(X_i, X_{i+k}) \in E \supset k > 0$), then we can use this conditional independence property to simplify our expression for the joint distribution:

$$\Pr(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} \Pr(X_i|\text{Pa}(X_i)). \qquad (7.5)$$

The nice thing about Equation 7.5 is that the product terms [illegible]he right-hand side are exactly the marginal and conditional probabilities required to quantify the network.

In using probabilistic networks for planning and control, we generally wish to compute the posterior distribution for some random variable given some evidence o some proposed action or contemplated observation. Intuitively, we are 'nterested in updating our beliefs given the evidence obtained so far, and reasoning hypothetically about possible future courses of action. In the vacation trip example, before we start out on the trip, we hypothesize about taking various routes and making information gathering side trips. After stopping at the state police station, we update our beliefs regarding the status of the bridge by incorporating the evidence obtained from the police.

To capture this process of updating beliefs and reasoning hypothetically, we introduce the notion of a *belief function* defined on each of the random variables in $\mathcal{G}$ as

$$\text{Bel}(X) = \Pr(X|\mathcal{E}),$$

where $\mathcal{E}$ represents all of the evidence obtained so far. Whenever we obtain new evidence, we extend $\mathcal{E}$ and update Bel(X) for all X of interest. Hypothetical reasoning is handled by including additional conditioning information, as in

$$\text{Bel}(X|Y) = \Pr(X|Y,\mathcal{E}).$$

We can compute Bel(X) directly using the joint distribution defined in Equation 7.5. For instance, suppose that $V = \{A, B, C\}$, and we have obtained as evidence the actual value of B. To compute the belief function on A given the evidence regarding B, we need $\Pr(A|B)$. By the definition of conditional probability, we have

$$\Pr(A|B) = \frac{\Pr(A,B)}{\Pr(A)}.$$

We can obtain $\Pr(A)$ by summing the joint distribution over all variables except A as in

$$\Pr(A) = \sum_{c\in\Omega_C} \sum_{b\in\Omega_B} \Pr(A, B = b, C = c).$$

This is referred to as *marginalizing* the joint probability distribution to A. We obtain $\Pr(A, B)$ in a similar manner as

$$\Pr(A, B) = \sum_{c\in\Omega_C} \Pr(A, B, C = c).$$

This particular method of computing belief functions can involve a number of arithmetic operations linear in the size of the joint probability space:

$$\prod_{X\in V} |\Omega_X|.$$

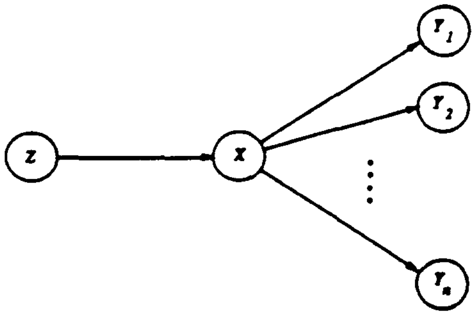


Figure 7.7: Simple tree-structured probabilistic network

In many cases, we can do significantly better from a computational standpoint by exploiting the structure of the graph. In particular, if $\mathcal{G}$ is a tree (*i.e.*, for all $v \in V$, $\text{Pa}(v) \leq 1$), then we can compute the belief function for all variables in V in time proportional to

$$\sum_{X \in V} \left(\prod_{Y \in \text{Pa}(X)} |\Omega_Y| \right).$$

For trees, the only information required to compute the belief function at a given node can be obtained from adjacent nodes in the graph. The complexity arises from the local structure of the graph.

In the following, we describe how to compute the belief function for trees. While trees occur infrequently in practice, the exercise provides some additional insight into probabilistic networks. Following the description of the method for handling trees, we describe a method of transforming arbitrary probabilistic networks into hyper graphs with tree-like structure that can be handled by methods similar to those used for trees.

Consider how we might compute $\Pr(X|Z, Y_1, \ldots, Y_n)$ given the tree-structured probabilistic network shown in Figure 7.7. Applying Bayes rule, we have

$$\Pr(X|Z, Y_1, \ldots, Y_n) = \frac{\Pr(Z, Y_1, \ldots, Y_n|X)\Pr(X)}{\Pr(Z, Y_1, \ldots, Y_n)}.$$

Marginalizing in the denominator, we have

$$\Pr(Z, Y_1, \ldots, Y_n) = \sum_{x \in \Omega_X} \Pr(Z, Y_1, \ldots, Y_n|X)\Pr(X).$$

Using conditional independence and applying Bayes rule again, we have

$$\Pr(Z, Y_1, \ldots, Y_n | X) = \frac{\Pr(X|Z)\Pr(Z)\Pr(Y_1|X)\cdots\Pr(Y_n|X)}{\Pr(X)}.$$

Substituting, we have

$$\Pr(X|Z, Y_1, \ldots, Y_n) = \frac{\Pr(X|Z)\Pr(Z)\Pr(Y_1|X)\cdots\Pr(Y_n|X)}{\sum_{x\in\Omega_X}\Pr(X = x|Z)\Pr(Z)\Pr(Y_1|X = x)\cdots\Pr(Y_n|X = x)},$$

which requires only the marginal and conditional probabilities necessary to quantify the probabilistic network shown in Figure 7.7.

For the problems we will be considering, evidence corresponds to the instantiation of variables at the *boundary* of the network (*i.e.*, variables with no parents or no children). The impact of evidence on variables not on the boundary has to be assessed by propagating the effects of evidence through intervening variables. In Figure 7.7, the set $\{Z, Y_1, \ldots, Y_n\}$ corresponds to the boundary. Some or all of the variables in the boundary may be instantiated in response to observations made by the agent. After each observation, the *belief function will require updating*. For instance, having determined $\Pr(X|Z, Y_1, \ldots, Y_n)$, we can compute $\text{Bel}(X)$ for $\mathcal{E} = Z = z, Y_1 = y_1, \ldots, Y_n = y_n$.

Let **e** represent all of the evidence obtained thus far. Removing X separates $\mathcal{G}$ into $n+1$ subtrees associated with the single parent of X and its n children. We partition **e** into $n+1$ components corresponding to these $n+1$ subtrees. Let $\mathbf{e}^+$ be the evidence associated with the parent of X, and $\mathbf{e}_i^-$ be the evidence associated with the ith child of X. Figure 7.8 illustrates this partition graphically. Suppose that X can obtain $\Pr(Z|\mathbf{e}^+)$ from Z, and $\Pr(\mathbf{e}_i^-|Y_i)$ from Y_i. Given this information, we can compute $\Pr(X|\mathbf{e})$ in a manner similar to that used in computing $\Pr(X|Z, Y_1, \ldots, Y_n)$ above:

$$\Pr(X|\mathbf{e}) = \Pr(X|\mathbf{e}^+, \mathbf{e}_1^-, \ldots, \mathbf{e}_n^-) = \frac{\Pr(X|\mathbf{e}^+)\Pr(\mathbf{e}_1^-|X)\cdots\Pr(\mathbf{e}_n^-|X)}{\sum_{x\in\Omega_X}\Pr(X = x|\mathbf{e}^+)\Pr(\mathbf{e}_1^-|X = x)\cdots\Pr(\mathbf{e}_n^-|X = x)},$$

where $\Pr(X|\mathbf{e}^+)$ is obtained from $\Pr(X|Z)$ and $\Pr(Z|\mathbf{e}^+)$ as follows:

$$\Pr(X|\mathbf{e}^+) = \sum_{z\in\Omega_Z}\Pr(X|Z = z)\Pr(Z = z|\mathbf{e}^+).$$

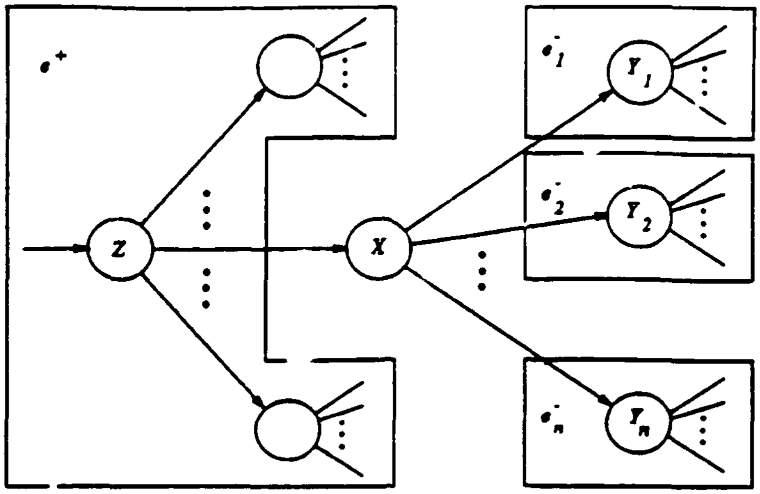


Figure 7.8: Partitioning the evidence bearing on X into subtrees

Evidence propagation occurs by local message passing. Each node keeps track of $n+1$ messages corresponding to the last messages received from its single parent and each of its n children. The node corresponding to X recomputes $\Pr(X|\mathbf{e})$ only in the event that it receives a message from a parent or child that differs from the last message received from that same parent or child. Nodes corresponding to evidence ignore incoming messages. If X recomputes $Pr(X|\mathbf{e})$, it also recomputes appropriate messages to send to its parent and children, and then sends these messages. The message X sends to its parent Z is computed as

$$\Pr(\mathbf{e}_X^-|Z) = \sum_{x\in\Omega_X}\left(\Pr(X = x|Z)\prod_{i=1}^{n}\Pr(\mathbf{e}_i^-|X)\right),$$

where $\mathbf{e}_X^-$ indicates the evidence in the subtree rooted at X. The message X sends to its kth child is computed as

$$\Pr(X|\mathbf{e}_{X/Y_k}^+) = \Pr(X|\mathbf{e}^+, \mathbf{e}_1^-, \ldots, \mathbf{e}_{k-1}^-, \mathbf{e}_{k+1}^-, \ldots, \mathbf{e}_n^-).$$

where $\mathbf{e}_{X/Y_k}^+$ indicates all of the evidence in the tree rooted at X except that found in the subtree rooted at Y_k. Note that the right-hand side of this expression can be computed in a manner similar to that used in computing $\Pr(X|\mathbf{e})$ by simply eliminating the $\Pr(\mathbf{e}_k^-|X)$ factor.

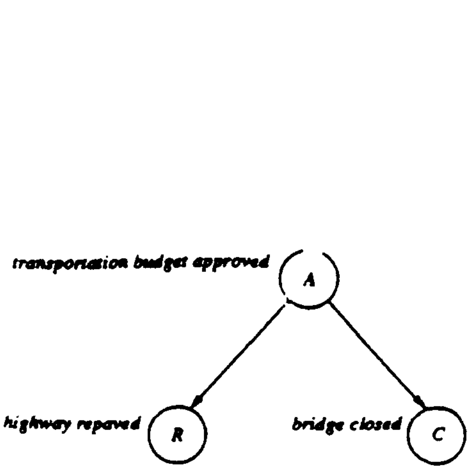


Figure 7.9: Propagating evidence in trees

All of these messages require only information available from the either originating node or from messages sent by parents and children. If we assume unit cost for updating the local information at stored at a node in response to a new message, then the cost of updating Bel(X) for all $X \in V$ in response to new evidence originating at a single node is proportional to V in the worst case.

Consider the following example illustrating how evidence propagates through a tree-structured network. The example extends the earlier example concerning the status of a critical bridge in planning a vacation trip. Suppose that we know some additional information regarding the status of this bridge. In particular, suppose we know that the repairs to the bridge that would result in its closing are contingent upon an increase in the state transportation budget. This budget increase was to be voted on in the state legislature earlier in the year. Unfortunately, we did not hear the outcome of the vote, but the same increase was to be used to repave a portion of the highway that will have to be traversed near the beginning of the trip.

We introduce three boolean-valued random variables: A representing the proposition that the budget increase was *approved*, C representing the proposition that the bridge is *closed*, and R representing the proposition that the highway portion in question was *repaved*. Suppose that we have a prior distribution on the budget approval,

	Pr(A)
A = *true*	0.1
A = *false*	0.9

a conditional probability distribution for the bridge being closed given that the budget is approved,

	Pr(C|A)	
	A = true	A = false
C = true	0.7	0.2
C = false	0.3	0.8

and a conditional probability distribution for the highway being repaved given that the budget is approved.

	Pr(R|A)	
	A = true	A = false
R = true	0.6	0.1
R = false	0.4	0.9

The resulting network is shown in Figure 7.9. Now. suppose that during the early part of our trip we discover that the portion of the highway in question has indeed been repaved. We want to update the network to reflect the evidence: $R = true$. For the purposes of the trip example, we are interested in

$$\text{Bel}(C = true) = \Pr(C = true|R = true)$$

in order to determine whether to take the direct or detour routes. To update C. we will also update A in the process of propagating the impact of the evidence.

For the simple network shown in Figure 7.9, we can easily compute the belief function using the joint distribution,

$$\Pr(A, C, R) = \Pr(C|A)\Pr(R|A)\Pr(A),$$

As described earlier, by definition we have

$$\Pr(C = true|R = true) = \frac{\Pr(C = true. R = true)}{\Pr(R = true)}.$$

Marginalizing, we compute the numerator by summing over Ω_A.

$$\begin{aligned}&\Pr(C = true. R = true) = \\ &\quad \Pr(C = true|A = true)\Pr(R = true|A = true)\Pr(A = true) + \\ &\quad \Pr(C = true|A = false)\Pr(R = true|A = false)\Pr(A = false),\end{aligned}$$

and the denominator by summing over $\Omega_A \times \Omega_C$.

$$\Pr(R = true) =$$

$$\begin{aligned}
&\Pr(C = \mathit{true} | A = \mathit{true}) \Pr(R = \mathit{true} | A = \mathit{true}) \Pr(A = \mathit{true}) + \\
&\Pr(C = \mathit{true} | A = \mathit{false}) \Pr(R = \mathit{true} | A = \mathit{false}) \Pr(A = \mathit{false}) + \\
&\Pr(C = \mathit{false} | A = \mathit{true}) \Pr(R = \mathit{true} | A = \mathit{true}) \Pr(A = \mathit{true}) + \\
&\Pr(C = \mathit{false} | A = \mathit{false}) \Pr(R = \mathit{true} | A = \mathit{false}) \Pr(A = \mathit{false}).
\end{aligned}$$

to obtain the value 0.4 for $\Pr(C = \mathit{true} | R = \mathit{true})$. Now, consider how we might obtain the same value by local message passing.

Prior to obtaining any evidence A, just sends $Pr(A)$ to C and R, and C and R send the function that maps all of Ω_C to 1.0. After C updates itself, we have $\Pr(C = \mathit{true}) = 0.25$. After obtaining the evidence $R = \mathit{true}$, R computes

$$\begin{aligned}
\Pr(R = \mathit{true} | A) \quad = \quad & \Pr(R = \mathit{true} | R = \mathit{true}) \Pr(R = \mathit{true} | A) + \\
& \Pr(R = \mathit{true} | R = \mathit{false}) \Pr(R = \mathit{false} | A),
\end{aligned}$$

and sends this message to A.

In response to this message, A updates its belief using the new message from R and the old one from C:

$$\Pr(A | R = \mathit{true}) = \frac{\Pr(A) \Pr(R = \mathit{true} | A)}{\Pr(A = \mathit{true}) \Pr(R = \mathit{true} | A = \mathit{true}) + \Pr(A = \mathit{false}) \Pr(R = \mathit{true} | A = \mathit{false})}.$$

The message sent to C from A is just $\Pr(A | R = \mathit{true})$, and C updates its belief as

$$\begin{aligned}
\Pr(C | R = \mathit{true}) \quad = \quad & \Pr(C | A = \mathit{true}) \Pr(A = \mathit{true} | R = \mathit{true}) + \\
& \Pr(C | A = \mathit{false}) \Pr(A = \mathit{false} | R = \mathit{true}),
\end{aligned}$$

from which we compute $\Pr(C = \mathit{true} | R = \mathit{true}) = 0.4$, the same value obtained using the joint distribution.

It is fairly straightforward to extend the above method for evaluating probabilistic models to handle networks in which a given node has more than one parent, but there is at most one path between any two nodes in the corresponding undirected graph. Such networks are called *singly-connected*. The extension involves keeping track of the evidence originating from subgraphs associated with the nodes of parents. Since there is a one-to-one correspondence between the parents and children of a given node and the set of subgraphs resulting from removing that node, keeping track of

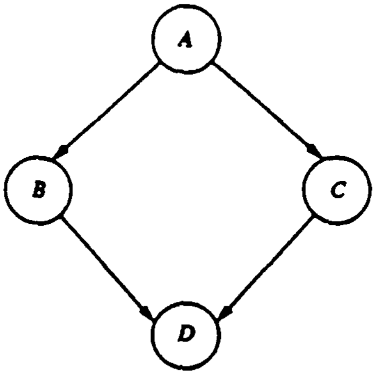


Figure 7.10: A multiply-connected network

evidence is relatively simple in singly-connected networks. The same cannot be said for *multiply-connected networks*, networks in which there are cycles in the corresponding undirected graph. Figure 7.10 shows a simple multiply-connected network. Problems arise in trying to distribute the impact of the evidence on A to B and C. In the worst case, correctly routing evidence about in a multiply-connected network requires a global perspective. Cooper has shown that exact evaluation of general probabilistic networks is NP-hard [9].

While computing the belief function for variables in probabilistic networks is intractable in the general case, we can often exploit the structure inherent in particular networks to reduce the cost of computation. One approach involves finding a set of variables, $\{X_1, \ldots, X_n\}$, which, if removed from the network, would render it singly connected (*e.g.*, the set $\{B\}$ in Figure 7.10). The belief function for a given node is taken as the weighted sum of the belief functions computed for all possible instantiations of the variables in $\{X_1, \ldots, X_n\}$. Calculating the weights is a rather complex, but the real trick involves finding a small set of variables to render the network singly connected. This is crucial since you have to calculate the belief function for $\prod_{i=1}^{n} |\Omega_{X_i}|$ variable instantiations.

A second approach to evaluating general probabilistic networks also involves converting multiply-connected networks into singly-connected ones. This approach involves constructing a hyper graph whose vertices correspond to the cliques of the chordal graph formed by triangulating the moral graph for the given network. [[Say a little more about triangulation and chordal graphs.]] From this hyper graph, we extract a maximal spanning

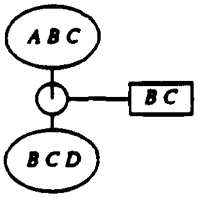


Figure 7.11: Join tree for a multiply-connected network

tree which is referred to as a *join tree*. [[Say a little more about maximal spanning trees.]] Figure 7.11 shows a join tree for the network of Figure 7.10.

$$\Pr(A,B,C,D) = \Pr(D|B,C)\Pr(B|A)\Pr(C|A)\Pr(A)$$

[[Say something about the messages passed in evaluating join trees. Provide some insight into Jensen's variation on Lauritzen and Spiegelhalter by updating the graph shown in Figure 7.11 (*e.g.*, the role of the running intersection property).]]

The cost of evaluating a probabilistic network using the join-tree approach is largely determined by the sizes of the state spaces formed by taking the cross product of the state spaces of the nodes in each vertex (clique) of the join tree. We can obtain an accurate estimate of the cost of evaluating a probabilistic network. $\mathcal{G} = (V, E)$ as follows. Let $C = \{C_i\}$ be the set of cliques in the chordal graph described earlier, where each clique represents a subset of V. We define the function, $\mathrm{Card} : C \rightarrow \{1, \ldots, |C| - 1\}$, so that $\mathrm{Card}(C_i)$ is the rank of the highest ranked node in C_i, where rank is determined by the maximal cardinality ordering of V. [[Say a little more about maximum cardinality ordering.]] We define the function, $\mathrm{Adj} : C \rightarrow 2^C$, by:

$$\mathrm{Adj}(C_i) = \{C_j | (C_j \neq C_i) \wedge (C_i \cap C_j \neq \emptyset)\}.$$

The join tree for G is constructed as follows. Each clique $C_i \in C$ is connected to the clique C_j in $\mathrm{Adj}(C_i)$ that has lower rank by $\mathrm{Card}(.)$ and has the highest number of nodes in common with C_i (ties are broken arbitrarily). Whenever we connect two cliques C_i and C_j, we create the *separation set* $S_{ij} = C_i \cap C_j$. The set of separation sets S is all the S_{ij}'s. We define the function, $\mathrm{Sep} : C \rightarrow 2^S$, by:

$$\mathrm{Sep}(C_i) = \{S_{jk} | S_{jk} \in S, (j = i) \vee (k = i)\}.$$

Finally, we define the *join-tree cost* as

$$\sum_{C_i \in C} \left(|\mathrm{Sep}(C_i)| \prod_{n \in C_i} |\Omega_n| \right).$$

where Ω_n is the state space of node n.

Say something about the multiply-connected case. Given that the subsequent sections will refer to Jensen's variation on Lauritzen and Spiegelhalter, that algorithm should be described as some level deeper than already attempted. An extremely detailed description is probably not warranted given that the material is readily available in a number of recent textbooks (e.g., [32, 31]).

Introduce influence diagrams and relate them to the decision trees described earlier in the introductory sections.

The first two examples of applying Bayesian networks to planning and control problems come from [11]. The first example considers the relatively simple problem of recognizing locally distinctive places. The second example considers the problem of choosing between paths through known and unknown territory. The latter example can be used to illustrate some of the tradeoff involved in working with multiply connected networks.

7.3 Robot Navigation

A significant problem in designing mobile robot control systems involves coping with the uncertainty that arises in moving about in an unknown or partially unknown environment and relying on noisy or ambiguous sensor data to acquire knowledge about that environment. In this section, we consider a control system that chooses what activity to engage in next on the basis of expectations about how the information returned as a result of a given activity will improve its knowledge about the spatial layout of its environment. Certain of the higher-level components of the control system are specified in terms of probabilistic decision models whose output is used to mediate the behavior of lower-level control components responsible for movement and sensing. The objective it to design control systems capable of directing the behavior of a mobile robot in the exploration and mapping of its environment, while attending to the real-time requirements of navigation and obstacle avoidance.

We are interested in building systems that construct and maintain representations of their environment for tasks involving navigation. Such systems should expend effort on the construction and maintenance of these representations commensurate with expectations about their value for immediate and anticipated tasks. Such systems should employ expectations about the information returned from sensors to assist in choosing activities that are most likely to improve the accuracy of its representations. Finally, in addition to reasoning about the future consequences of acting, such systems must attend to the immediate consequences of acting in a changing environment: consequences that generally cannot be anticipated and hence require some amount of continuous attention and commitment in terms of computational resources.

We start with the premise that having a map of your environment is generally a good thing if you need to move between specific places whose locations are clearly indicated on that map. The more frequent your need to move between locations, the more useful you will probably find a good map. If you are not supplied with a map and you find yourself spending an inordinate amount of time blundering about, it might occur to you to build one, but the amount of time you spend in building a map will probably depend upon how much you anticipate using it. Once you have decided to build a map, you will have to decide when and exactly how to go about building it. Suppose that you are on an errand to deliver a package and you know of two possible routes, one of which is guaranteed to take you to your destination and a second which is not. By trying the second route, you may learn something new about your environment that may turn out to be useful later, but you may also delay the completion of your errand.

The mobile robot that we consider in the examples in the rest of this chapter is a simple holonomic (turn-in-place) robot equipped with a number of sensors. The most important sensor for our immediate purposes is the ultrasonic sonar sensor considered in the previous chapter. The robot's ultrasonic sensors provide it with information about the distance to nearby objects. With a little care, the robot can detect the presence of a variety of geometric features using these sensors. In gathering information about the office environment, the robot will drive up to a surface to be investigated, align one of the sensors to the right or to the left of its direction of travel along the surface, and then move parallel to that surface looking for abrupt changes in the information returned by the aligned sensor that would indicate some geometric feature such as a 90° corner. In doing this, it is possible to keep track of the accumulated error in its movement and the variation in its sensor data to assign a probability to whether or not a feature is present.

We assume that the robot has strategies for checking out many simple geometric features found in typical office environments: we refer to these strategies as *feature detectors*. Each feature detector is realized as a control process that directs the robot's movement and sensing. On the basis of the data gathered during the execution of a given feature detector, a probability distribution is determined for the random variable corresponding to the proposition that the feature is present at a specific location.

The robot that we consider here is designed to explore its environment in order to build up a representation of that environment suitable for route planning. In the course of exploration, the robot induces a graph that captures certain qualitative features of its environment. In addition to detecting geometric features like corners and door jambs, the robot is able to classify locations. In particular, it is able to distinguish between corridors and places where corridors meet or are punctuated by doors leading to offices, labs, and storerooms. A corridor is defined as a piece of rectangular space bounded on two sides by uninterrupted parallel surfaces 1.5 to 2 meters apart and bounded on the other two sides by *ports* indicated by abrupt changes in one of the two parallel surfaces. The ports signal *locally distinctive places* (LDPs) (after [23]) which generally correspond to hallway junctions. Uninterrupted corridors are represented as arcs in the induced graph while junctions are represented as vertices. Junctions are further partitioned into classes of junctions (*e.g.*, L-shaped junctions where two corridors meet at right angles, or T-shaped junctions where one corridor is interrupted by a second perpendicular corridor). We will assume that the robot is given a set of junction classes that it uses to classify and the label the locations encountered during exploration.

In the following sections, we consider two of the main decision processes that comprise the robot's control system, but first we consider briefly the overall architecture in which these decision processes are embedded.

In the following, we assume a multi-level control system composed of a set of decision processes running concurrently under a multi-tasking prioritized operating system. There is no shared state information; all communication is handled by inter-process message passing. Run-time process arbitration is handled by dynamically altering the process priorities. Coordination among processes is achieved through a set of message-passing protocols.

The different processes that make up the controller are partitioned into levels (see Figure 7.12). For each level, there is a corresponding arbitrator designed to coordinate the different processes located at that level. At Level 0, we find the processes responsible for control of the different sensor/effector systems on board the mobile base. Each Level 0 process is

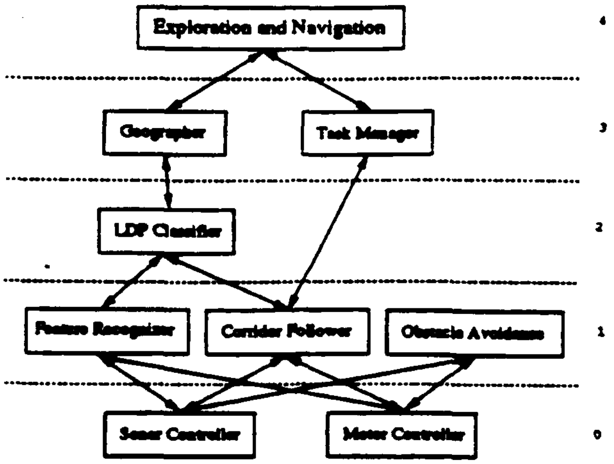


Figure 7.12: Mobile robot control architecture

completely independent of the other processes, so no arbitration is needed. At Level 1, we find the processes responsible for the low-level control of the robot. Level 1 processes are coordinated using a simple priority scheme: the obstacle avoidance process always takes priority over the other Level 1 processes. The activities of the feature recognition and corridor following processes are coordinated by higher-level processes.

In the design shown in Figure 7.12, there is only one Level 2 process, the LDP classifier, but, in a more complicated architecture, one could easily imagine several processes on this level. At Level 3, we find the two processes responsible for the robot's higher-level behaviors: the task manager in charge of running user-specified errands, and the geographer in charge of exploration and map building. Both the geographer and the task manager are special-purpose route planners: the geographer tends to construct paths through unknown territory and the task manager through known territory. The activities of these two processes are coordinated by a Level 4 decision process that takes into account the possible costs and benefits to be derived from different strategies for mixing exploration and errand running. In the following, we consider the decision processes at Levels 2 and 4.

7.3.1 Classifying Locally Distinctive Places

Upon exiting a corridor through a port, the robot will want to determine what sort of LDP it has entered. If the robot is in a well-explored portion of its environment. this determination should match its expectations as indicated in its m p. If, on the other hand, the robot is in some unknown or only partially-explored area, this determination will be used to extend the map, possibly adding new vertices or identifying the current LDP with existing vertices. In this section, we describe how the robot might classify LDPs encountered during exploration.

Let L be the set of all locally distinctive places in the robot's environment, $C = \{C_1, C_2, \ldots, C_n\}$ be a set of equivalence classes that partitions L, and F be a set of primitive geometric features (*e.g.*, convex and concave corners, flat walls). Each class in C can be characterized as a set of features in F that stand in some spatial relationship to one another. As the robot exits a port, a local coordinate system is set up with its origin on the imaginary line defined by the exit port and centered in the corridor. The space about the origin enclosing the LDP is divided into a set of equi-angular wedges W. For each feature/wedge pair (f, w) in $F \times W$, we define a specialized feature detector $d_{f,w}$ that is used to determine if the current LDP satisfies the feature f at location w in the coordinate system established upon entering the LDP. Let D be the set of all such feature detectors plus *no_op*, a pseudo-detector that results in no new information and takes no time or effort to execute.

The LDP-classification module maintains a a probabilistic assessment of the hypotheses concerning the class of the current LDP given the evidence acquired thus far. At any given time, the robot will have tried some number of feature detectors. Let P_t be the pool of detectors available for use at time t; P_t is just D less the set of detectors executed up until t in classifying the current LDP. The LDP-classification module is responsible for choosing the next feature detector to invoke from the set P_t. It does so using a decision model cast in terms of an influence diagram.

The LDP-classification module's influence diagram includes a set of chance nodes corresponding to random variables, a decision node corresponding to actions that the robot might take, and a value node representing the expected utility of invoking the different feature detectors in various circumstances. The chance nodes include a hypothesis variable, H, that can take on values from C, and a set of boolean variables of the form, $X_{f,w}$, used to represent whether or not the feature f is present at location w. Each $X_{f,w}$ is conditioned on the hypothesis H according to the distribution $\Pr(X_{f,w}|C_i)$ determined by whether or not the class requires the feature at the specified location. The decision node, P_t, indicates the feature detectors available for use at time t, and the value node, V, represents the utility of invoking each

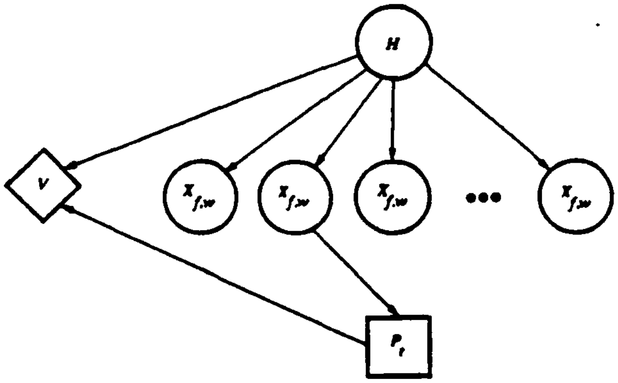


Figure 7.13: LDP-classification module's influence diagram

feature detector. V is dependent only upon the hypothesis and decision nodes. The predecessors of P_t are just the feature detectors invoked so far, thereby indicating temporal precedence and informational dependence. A graphical representation of the influence diagram is shown in Figure 7.13.

The utility of invoking each detector is based on (i) the ability of the detector to discriminate among the hypotheses, (ii) the cost of deploying the detector, (iii) the probability that the current best hypothesis is correct, and (iv) the cost of misidentifying the LDP. The first two are used to select from among $D - \{no\_op\}$ and the last two are used to choose between the best detector from $D - \{no\_op\}$ and no_op. The LDP-classification module selects from $D - \{no\_op\}$, using the function, $\mu : P_t \times H \rightarrow \Re$, defined by $\mu(d_{f,w}, h) =$

$$\kappa_1 \mathrm{Discrim}(d_{f,w}) - \kappa_2 \mathrm{Cost}(d_{f,w}, h),$$

where κ_1 and κ_2 are constants used for scaling, $\mathrm{Cost}(d_{f,w}, h)$ is a function of the expected time spent in executing $d_{f,w}$ for an LDP of a given class, and $\mathrm{Discrim}(d_{f,w})$ is a variation on a standard discrimination function used in pattern recognition, and defined by

$$\sum_{i=1}^{n} \Pr(C_i) \sum_{v \in \{0,1\}} |\Pr(d_{f,w} = v|C_i) - \Pr(d_{f,w} = v)|,$$

where $d_{f,w} = v$ is meant to represent the proposition that the detector $d_{f,w}$ returns the value v. The terms in the above formula are easily obtained. $\Pr(d_{f,w} = v|C_i)$ is the distribution associated with the corresponding $X_{f,w}$

node. and $\Pr(d_{f,w} = v)$ can be calculated using

$$\Pr(d_{f,w} = v) = \sum_{i=1}^{n} \Pr(d_{f,w} = v|C_i)\Pr(C_i)$$

The LDP-classification module evaluates the influence diagram using one of the methods described in Section 7.2 to obtain a decision policy and an expected value function for choosing from among $D - \{no\_op\}$. The LDP-classification module can also choose to do nothing by selecting no_op, thereby committing to the class C_i with the highest posterior probability given the information returned by the feature detectors invoked thus far. In a more realistic decision model, we might employ an additional set of chance nodes corresponding to micro features and a more extensive the set of feature than indicated here. We would also want to allow for a feature detector to be invoked multiple times.

7.3.2 Expected Value of Exploration

One could imagine several decision models for reasoning about the expected value of exploration. In the simple model presented in this section, we assume that the system of junctions and corridors that make up the robot's environment can be registered on a grid so that every corridor is aligned with a grid line and every junction is coincident with the intersection of two grid lines. In the following, the set of junction types, J, corresponds to all possible configurations of corridors incident on the intersection of two grid lines. Intersections with at least one incident corridor correspond to LDPs. Since we also assume that the robot knows the dimensions of the grid (*i.e.*, the number of x and y grid lines), we can enumerate the set of possible maps $M = \{M_1, M_2, \ldots, M_m\}$, where a map corresponds to an assignment of a junction type to each intersection of grid lines. For most purposes, we can think of a map as a labeled graph.

We can restrict M by making a number of assumptions about office buildings of the sort that the robot will find itself in (*e.g.*, all LDPs are connected). To further restrict M, the robot engages in an initial phase of task-driven exploration. Each task specifies a destination location in x, y grid coordinates. The robot computes the shortest path assuming that all intersections have as many coincident corridors as is consistent with what is known about the intersection and its adjacent intersections. The robot then follows this path, acquiring additional information as it moves through

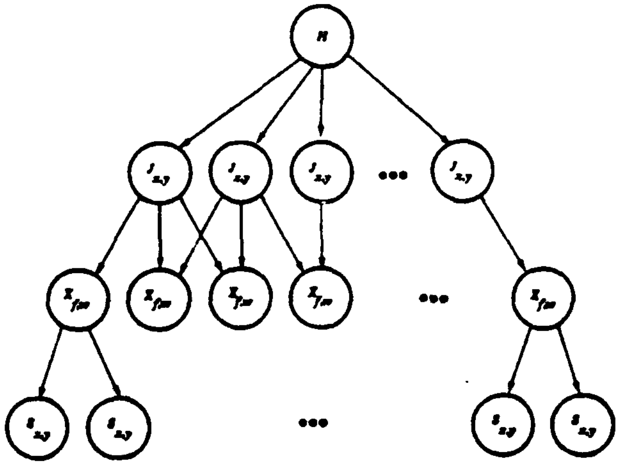


Figure 7.14: The probabilistic model for map building

unknown intersections until it either finds its path blocked, in which case it recomputes the shortest path to the goal taking into account its new knowledge, or it reaches the goal.

The robot continues in this task-driven exploration phase until it is likely—based on the spatial distribution of known locations—that all locations have been visited at least once. From this point on, given a task to move to specific location, it is likely that it will be able to compute a path through known territory. The robot now faces the decision whether to take the known path or to try an alternative path through unknown territory. In the model considered here, the robot has to choose between taking the shortest path through known territory, and trying the shortest path consistent with what is known. In the latter case, it will learn something new, but it may end up taking longer to complete its task.

Let H be a random variable corresponding to the actual configuration of the environment; H takes on values from M. Let $J_{x,y}$ be a random variable corresponding to the junction type of the intersection at the coordinates, $\langle x, y \rangle$, in the grid; $J_{x,y}$ can take on values from the set C defined previously. Let $X_{f,m}$ be as previously defined, a boolean variable corresponding the presence of a feature at a particular position. Let $S_{x,y}$ be a random variable corresponding to a possible sensing action taken at the coordinates, $\langle x, y \rangle$, in the grid. Let $\mathcal{E}$ correspond to the set of sensing actions taken thus far. The complete probabilistic model is shown in Figure 7.14.

In our simple model, the robot has to decide between the two alternatives, P_K and P_U, corresponding to paths through known and unknown territory. To compute $\Pr(H|\mathcal{E})$, $\Pr(H)$ is assumed to be uniform, $\Pr(J_{x,y}|H)$ and $\Pr(X_{f,w}|J_{x,y})$ are determined by the geometry, and $\Pr(S_{x,y}|X_{f,w})$ is determined experimentally. Let $T = \{T_1, T_2, \ldots, T_r\}$ denote the set of all tasks corresponding to point-to-point traversals, and $\mathrm{E}(|T_i|)$ denote the expected number of tasks of type T_i. Let $\mathrm{Cost}(T_i, M_j, M_k)$ be the time required for the task T_i using the map M_j, given that the actual configuration of the environment is M_k; if M_j is a subgraph of M_k, then $\mathrm{Cost}(T_i, M_j, M_k)$ is just the length of the shortest path in M_j. Let T^* denote the robot's current task. For evaluation purposes, we assume that the robot will take at most one additional exploratory step.

To complete the decision model, we need a means of computing the expected value of P_K and P_U. In general, the value of a given action is the sum of the immediate costs related to T^* and the costs for expected future tasks. Let

$$\mathrm{Futures}(M_i, I) = \sum_{j=1}^{r} \mathrm{E}(|T_j|)\,\mathrm{Cost}(T_j, M_I^*, M_i),$$

where $M_I^* = M_{\arg\max_j \Pr(M_j|I)}$.

If classification is perfect, the robot correctly classifies any location it passes through, and $M_{\mathcal{E}}^*$ is the minimal assignment consistent with what it has classified so far. In this case, the expected value of P_K is

$$\mathrm{Cost}(T^*, M^*, \_) + \mathrm{Futures}(\_, \mathcal{E}).$$

If classification is imperfect, the expected value of P_K is

$$\sum_{j=1}^{m} \Pr(M_j|\mathcal{E})\,[\mathrm{Cost}(T^*, M^*, M_j) + \mathrm{Futures}(M_j, \mathcal{E})].$$

Handling P_U is just a bit more complicated. Suppose that the robot is contemplating exactly one sensing action that will result in one of several possible observations $O_1, \ldots, O_n$, then the expected value of P_U is

$$\sum_{j=1}^{m} \Pr(M_j|\mathcal{E})\,\mathrm{Cost}(T^{*\prime}, M^*, M_j) +$$
$$\sum_{i=1}^{n} \Pr(O_i) \sum_{j=1}^{m} \Pr(M_j|O_i, \mathcal{E})\,\mathrm{Futures}(M_j, [O_i, \mathcal{E}])$$

where $T^{*'}$ is a modification of T^* that accounts for the proposed exploratory sensing action.

We use Jensen's [21] variation on Lauritzen and Spiegelhalter's [25] algorithm to evaluate the network shown in Figure 7.14. The time required for evaluation is determined by the size of the sample spaces for the individual random variables and the connectivity of the network used to specify the decision model. In the case of a singly-connected network, the cost of computation is polynomial in the number of nodes and the size of the largest sample space—generally the space of possible maps. The network shown in Figure 7.14 would be singly-connected if each feature, $X_{f,w}$, had at most one parent corresponding to a junction, $J_{x,y}$; a network of this form with *100* possible maps can be evaluated in about *10 seconds*, assuming an 8×8 grid.

In the case of a multiply-connected network, the cost of computation is a function of the product of the sizes of the sample spaces for the nodes in the largest clique of the graph formed by triangulating the DAG corresponding to the original network. By making use of the information gathered in the initial exploratory phase, the robot is able to reduce the connectivity of the network used to encode the decision model. Multiply-connected networks accounting for approximately 50 possible maps require on the order of a few minutes to evaluate.

The space of possible maps chosen may not include the map corresponding to the actual configuration of the environment. To handle such possible omissions, we add a special value, $\perp$, to the sample space for H, and make all of the $\Pr(J_{x,y}|\perp)$ entries in the conditional probability tables $1/s$ where s is the number of junction types. If the robot ever detects that $M_{\xi}^{*} = \perp$, then it assumes that it has excluded the real map, and dynamically adjusts its decision model by computing a new sample space for H guided by the results of the exploratory actions taken thus far.

7.3.3 Designing Robot Control Systems

One approach to designing control systems employing a decision-theoretic perspective is described as follows. We begin by considering the overall decision problem, determining an optimal decision procedure according to a precisely stated decision-theoretic criteria, neglecting computational costs. We use an influence diagram to represent the underlying decision model and define the optimal procedure in terms of evaluating this model.

In the case described above, the robot's overall decision problem involves several component problems associated with specific classes of events occurring in the environment. These component decision problems include what action to take when approached by an unexpected object in a corridor, what sensor action to take next when classifying a junction, and what path to take in combining exploration and task execution. Each of these problems is recurrent.

Problems involving what sensor action to take in classification or what path to take in navigation are predictably recurrent. For instance, during classification each sensor action takes about thirty seconds to a minute, so the robot has that amount of time to decide what the next action should be if it wishes to avoid standing idle lost in computation. The frequency with which choices concerning what path to take occur is dependent on how long the robot takes to traverse the corridor on route to the next LDP. With the current mobile platform operating in the halls of the computer science department, moving between two consecutive LDPs takes about four minutes. The problem of deciding what to do when approached by an unexpected object occurs unpredictably, and the time between when the approaching object is detected and when the robot must react to avoid a collision is on the order of a few seconds.

By making various (in)dependence assumptions and eliminating noncritical variables from the overall complex decision problem, we are able to decompose the globally optimal decision problem into sets of simpler component decision problems. Each of the sets of component problems are solved by a separate module. The computations carried out by these modules are optimized using a variety of techniques to take advantage of the expected time available for decision making. The different decision procedures communicate by passing probability distributions back and forth. For instance, the module responsible for making decisions regarding exploration and the module responsible for classifying LDPs pass back and forth distributions regarding the junction types of LDPs.

The control system described above combines high-level decision making with low-level control and sensor interpretation to provide for navigation, real-time obstacle avoidance, and exploration in an unfamiliar environment. The basic controller handles multiple asynchronous processes communicating via simple message-passing protocols. The architecture supports a variety of arbitration schemes from fixed-priority processor scheduling to decision-theoretic control. This section has emphasized two decision processes: one responsible for reasoning about the uncertainty inherent in dealing with noisy and ambiguous sensor data, and a second responsible for assessing the expected value of various exploratoˌy actions. Our basic approach to designing robot control systems invoˌves constructing a decision model for the overall problem and then decomposing it into component models guided by the time criticality of the associated decision problems.

The third example involves sequential decision making. and for this we have to introduce some additional machinery. In particular, the probabilistic projection approach described in [13. 15] and particularly [14]. Relate this to Tatman and Schacter's work on connecting influence diagrams and dynamic programming methods for sequential decision making involving Markov processes.

7.4 Change Over Time

Reasoning about change requires predicting how long a proposition, having become true. will continue to be so. Lacking perfect knowledge. an agent may be constrained to believe that a proposition persists indefinitely simply because there is no way for the agent to infer a contravening proposition with certainty. In this section, we describe a model of causal reasoning that accounts for knowledge concerning cause-and-effect relationships and knowledge concerning the tendency for propositions to persist or not as a function of time passing. The model has a natural encoding in the form of a network representation for probabilistic models. We will also consider how our probabilistic model addresses certain classical problems in temporal reasoning (*e.g.*, the frame and qualification problems).

The common-sense law o. inertia [27] states that a proposition once made true remains so until something makes it false. Given perfect knowledge of initial conditions and a complete predictive model. the law of inertia is sufficient for accurately inferring the persistence of propositions. In most circumstances, however, our predictive models and our knowledge of initial conditions are less than perfect. The law of inertia requires that. in order to infer that a proposition ceases to be true, we must predict an event with a contravening effect. Such predictions are often difficult to make. Consider the following examples:

- a cat is sleeping on the couch in your living room
- you leave your umbrella on the 8:15 commuter train
- a client on the telephone is asked to hold

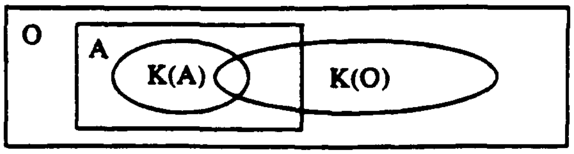


Figure 7.15: Events precipitate change in the world

In each case, there is some proposition initially observed to be true, and the task is to determine if it will be true at some later time. The cat may sleep undisturbed for an hour or more, but it is extremely unlikely to remain in the same spot for more than six hours. Your umbrella will probably not be sitting on the seat when you catch the train the next morning. The client will probably hold for a few minutes, but only the most determined of clients will be on the line after 15 minutes. Sometimes we can make more accurate predictions (*e.g.*, a large barking dog runs into the living room), but, lacking specific evidence, we would like past experience to provide an estimate of how long certain propositions are likely to persist.

Events precipitate change in the world, and it is our knowledge of events that enables us to make useful predictions about the future. For any proposition P that can hold in a situation, there are some number of general sorts of events (referred to as *event types*) that can affect P (*i.e.*, make P true or false). For any particular situation, there are some number of specific events (referred to as *event instances*) that occur. Let O correspond to the set of events that occur at time t, A correspond to that subset of O that affect P, $K(O)$ that subset of O known to occur at time t, and $K(A)$ that subset of A whose type is known to affect P. Figure 7.15 illustrates how these sets might relate to one another in a specific situation. In many cases, $K(O) \cap K(A)$ will be empty while A is not, and it may still be possible to provide a reasonable assessment of whether or not P ¿is true at t. In this section, we provide an account of how such assessments can be made probabilistically.

7.4.1 Prediction and Persistence

In the following, we distinguish between two kinds of propositions: propositions, traditionally referred to as *fluents* [28], which, if they become true, tend to persist without additional effort, and propositions, corresponding to the occurrence of events, which, if true at a point, tend to precipitate or trigger change in the world. Let $\langle P, t\rangle$ indicate that the fluent P is true at time t, and $\langle E,t\rangle$ indicate that an event of type E occurs at time t. We use the notation E_P to indicate an event corresponding to the fluent P becoming true.

Given our characterization of fluents as propositions that tend to persist, whether or not P is true at some time t may depend upon whether or not it was true at $t-\Delta$, where $\Delta > 0$. We can represent this dependency as follows:[1]

$$\begin{aligned}\Pr(\langle P,t\rangle) \;=\; & \Pr(\langle P,t\rangle|\langle P,t-\Delta\rangle)\Pr(\langle P,t-\Delta\rangle) + \\ & \Pr(\langle P,t\rangle|\neg\langle P,t-\Delta\rangle)\Pr(\neg\langle P,t-\Delta\rangle)\end{aligned} \tag{7.6}$$

where $\neg\langle P,t\rangle \equiv \langle\neg P,t\rangle$.

The conditional probabilities $\Pr(\langle P,t\rangle|\langle P,t-\Delta\rangle)$ and $\Pr(\langle P,t\rangle|\neg\langle P,t-\Delta\rangle)$ are related to the *survivor function* in classical queuing theory [35]. Survivor functions encode the changing expectation of a fluent remaining true over the course of time. We employ survivor functions to capture the tendency of propositions to become false as a consequence of events with contravening effects. With survivor functions, one need not be aware of a specific instance of an event with a contravening effect in order to predict that P will cease being true. As an example of a survivor function,

$$\Pr(\langle P,t\rangle) = e^{-\lambda\Delta}p(\langle P,t-\Delta\rangle)$$

indicates that the probability that P persists drops off as a function of the time since P was last observed to be true at an exponential rate determined by λ (Figure 7.16). The exponential decay survivor function is equivalent to the case where

$$\Pr(\langle P,t\rangle|\langle P,t-\Delta\rangle) = e^{-\lambda\Delta}$$

and

$$\Pr(\langle P,t\rangle|\neg\langle P,t-\Delta\rangle) = 0.$$

Referring back to Figure 7.15, survivor functions account for that subset of A corresponding to events that make P false, assuming that $K(A) = \{\}$.

[1] The equality in Formula 7.6 follows from the *generalized addition law*: if $A_1,\ldots,A_n$ are exclusive and exhaustive and B is any event, then

$$\Pr(B) = \sum_{i=1}^{n} \Pr(B|A_i)\Pr(A_i).$$

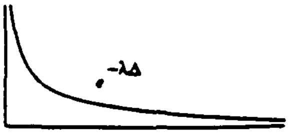


Figure 7.16: A survivor function with exponential decay

If we have evidence concerning specific events known to affect P (*i.e.*, $K(A) \cap K(O) \neq \{\}$), Formula 7.6 is inadequate. As an interesting special case of how to deal with events known to affect P, suppose that we know about all events that make P true (*i.e.*, we know $\Pr(\langle E_P, t\rangle)$ for any value of t), and none of the events that make P false. In particular, suppose that P corresponds to John being at the airport, and E_P corresponds to the arrival of John's flight. We are interested in whether or not John will still be waiting at the airport when we arrive to pick him up. Let $\sigma(t) = e^{-\lambda t}$ represent John's tendency to hang around airports, where λ is a measure of his impatience. If $f(t) = \Pr(\langle E_P, t\rangle)$, then we can compute the probability of P being true at t by convolving f with the survivor function σ as in

$$\Pr(\langle P, t\rangle) = \int_{-\infty}^{t} \Pr(\langle E_P, z\rangle)\sigma(t - z)dz \qquad (7.7)$$

A shortcoming of Formula 7.7 is that it fails to account for evidence concerning specific events known to make P false. Suppose, for instance, that $E_{\neg P}$ corresponds to Fred meeting John at the airport and giving him a ride to his hotel. In certain cases,

$$\Pr(\langle P, t\rangle) = \int_{-\infty}^{t} \Pr(\langle E_P, z\rangle)\sigma(t - z)\left[1 - \int_{z}^{t} \Pr(\langle E_{\neg P}, x\rangle)dx\right]dz \qquad (7.8)$$

provides a good approximation. Figure 7.17 illustrates the sort of inference licensed by Formula 7.8.

There are some potential problems with Formula 7.8. The survivor function σ was meant to account for all events that make P false, but Formula 7.8 counts one such event, John leaving the airport with Fred, twice: once in the survivor function and once in $\Pr(\langle E_{\neg P}, t\rangle)$. In certain cases, this can lead to significant errors (*e.g.*, Fred always picks up John at the airport). To combine the available evidence correctly, it will help if we distinguish the different sorts of knowledge that might be brought to bear on estimating whether or not P is true. We will also reinterpret the event type E_P to

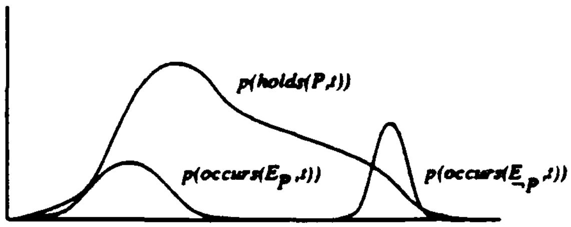


Figure 7.17: Probabilistic predictions

mean an event *known* to make P true. The following formula makes the necessary distinctions and indicates how the evidence should be combined:[2]

$$
\begin{array}{lll}
\Pr(\langle P,t\rangle) = & & (7.9) \\
\quad \Pr(\langle P,t\rangle | \langle P,t-\Delta\rangle \wedge \neg(\langle E_P,t\rangle \vee \langle E_{\neg P},t\rangle)) & (\mathrm{N1}) & \\
\quad\quad * \Pr(\langle P,t-\Delta\rangle \wedge \neg(\langle E_P,t\rangle \vee \langle E_{\neg P},t\rangle)) & & \\
+ \Pr(\langle P,t\rangle | \langle P,t-\Delta\rangle \wedge \langle E_P,t\rangle) & (\mathrm{N2}) & \\
\quad\quad * \Pr(\langle P,t-\Delta\rangle \wedge \langle E_P,t\rangle) & & \\
+ \Pr(\langle P,t\rangle | \langle P,t-\Delta\rangle \wedge \langle E_{\neg P},t\rangle) & (\mathrm{N3}) & \\
\quad\quad * \Pr(\langle P,t-\Delta\rangle \wedge \langle E_{\neg P},t\rangle) & & \\
+ \Pr(\langle P,t\rangle | \neg\langle P,t-\Delta\rangle \wedge \neg(\langle E_P,t\rangle \vee \langle E_{\neg P},t\rangle)) & (\mathrm{N4}) & \\
\quad\quad * \Pr(\neg\langle P,t-\Delta\rangle \wedge \neg(\langle E_P,t\rangle \vee \langle E_{\neg P},t\rangle)) & & \\
+ \Pr(\langle P,t\rangle | \neg\langle P,t-\Delta\rangle \wedge \langle E_P,t\rangle) & (\mathrm{N5}) & \\
\quad\quad * \Pr(\neg\langle P,t-\Delta\rangle \wedge \langle E_P,t\rangle) & & \\
+ \Pr(\langle P,t\rangle | \neg\langle P,t-\Delta\rangle \wedge \langle E_{\neg P},t\rangle) & (\mathrm{N6}) & \\
\quad\quad * \Pr(\neg\langle P,t-\Delta\rangle \wedge \langle E_{\neg P},t\rangle) & &
\end{array}
$$

Consider the contribution of the individual terms corresponding to the conditional probabilities labeled N1 through N6 in Formula 7.9. N1 accounts for *natural attrition*: the tendency for propositions to become false given no direct evidence of events known to affect P. N2 and N5 account for *causal accretion*: accumulating evidence for P due to events known to make P true. N2 and N5 are generally 1. N3 and N6, on the other hand, are generally 0, since evidence of $\neg P$ becoming true does little to convince us that P is true. Finally, N4 accounts for *spontaneous causation*: the tendency for propositions to suddenly become true with no direct evidence of events known to affect P.

[2] In order to justify our use of the generalized addition law in Formula 7.9, we assume that $\Pr(\langle E_P,t\rangle \wedge \langle E_{\neg P},t\rangle) = 0$ for all t.

By using a discrete approximation of time and fixing Δ, it is possible both to acquire the necessary values for the terms N1 through N6 and to use them in making useful predictions. If time is represented as the integers, and $\Delta = 1$, we note that the law of inertia applies in those situations in which the terms N1, N2, and N5 are always 1 and the other terms are always 0. In the rest of this section, we assume that time is discrete and linear and that the time separating any two consecutive time points is some constant δ. Only evidence concerning events *known* to make P true is brought to bear on $\Pr(\langle E_P, t\rangle)$. If $\Pr(\langle E_P, t\rangle)$ were used to summarize all evidence concerning events that make P true, then N1 would be 1.

7.4.2 Reasoning About Causation

Before we consider the issues involved in making predictions using knowledge concerning N1 through N6, we need to add to our theory some means of predicting additional events. We consider the case of one event causing another event. Deterministic theories of causation often use implication to model cause-and-effect relationships. For instance, to indicate that the occurrence of an event of type E_1 at time t causes the occurrence of an event of type E_2 following t by some $\delta > 0$ just in case the conjunction $P_1 \wedge P_2 \ldots \wedge P_n$ holds at t, we might write

$$(\langle P_1 \wedge P_2 \ldots \wedge P_n, t\rangle \wedge \langle E_1, t\rangle) \supset \langle E_2, t+\delta\rangle.$$

If the caused event is of a type E_P, this is often referred to as *persistence causation* [29]. In our model, the conditional probability

$$\Pr(\langle E_2, t+\delta\rangle | \langle P_1 \wedge P_2 \ldots \wedge P_n, t\rangle \ \wedge \ \langle E_1, t\rangle) = \pi$$

is used to indicate that, given an event of type E_1 occurs at time t, and P_1 through P_n are true at t, an event of type E_2 will occur following t by some $\delta > 0$ with probability π.

In moving to a probabilistic model of causation, there are some complications that we have to deal with. Consider, for example, the two rules:

$$(\langle P, t\rangle \wedge \langle E, t\rangle) \supset \langle E_R, t+\delta\rangle$$

and

$$(\langle P \wedge Q, t\rangle \wedge \langle E, t\rangle) \supset \langle E_R, t+\delta\rangle.$$

These two rules pose no problems for the deterministic theory of causation, since P and Q are either true or false, and the rules either apply or not.

In fact, the second rule is redundant However, in a probabilistic model. P and Q usually are not unambiguously true or false. Therefore, in the probabilistic causal theory consisting of

$$\Pr(\langle E_R, t+\delta\rangle | \langle P, t\rangle \wedge \langle E, t\rangle) = \pi_1$$

$$\Pr(\langle E_R, t+\delta\rangle | \langle P \wedge Q, t\rangle \wedge \langle E, t\rangle) = \pi_2$$

the second rule can no longer be considered redundant. Since the second rule is more specific than the first, it provides us with valuable additional information. In a complete account of the causes for E_R, we would also need

$$\Pr(\langle E_R, t+\delta\rangle | \langle P \wedge \neg Q, t\rangle \wedge \langle E, t\rangle) = \pi_3$$

and other information as well. Providing a complete account of the interactions among causes and between causes and their effects is important in modeling change in a probabilistic framework. In the following two sections, we will consider this issue in more detail.

7.4.3 An Example

The task in *probabilistic projection* is to assign each propositional variable of the form $\langle \varphi, t\rangle$ a certainty measure consistent with the constraints specified in a problem. In this section, we provide examples drawn from a simple factory domain that illustrate the sort of inference required in probabilistic projection. We begin by introducing some new event types:

Cl = "The mechanic on duty cleans up the shop"
As = "Fred tries to assemble Widget17 in Room101"

and fluents:

Wr = "The location of Wrench14 is Room101"
Sc = "The location of Screwdriver31 is Room101"
Wi = "Widget17 is completely assembled"

We assume that tools are occasionally displaced in a busy shop, and that Wr and Sc are both subject to an exponential persistence decay with a half life of one day; this determines N1 in Formula 7.9:

$$\begin{aligned}
\Pr(\langle Wr, t\rangle | \langle Wr, t-\Delta\rangle \wedge \neg(\langle E_{Wr}, t\rangle \vee \langle E_{\neg Wr}, t\rangle)) &= e^{-\lambda\Delta} \\
\Pr(\langle Sc, t\rangle | \langle Sc, t-\Delta\rangle \wedge \neg(\langle E_{Sc}, t\rangle \vee \langle E_{\neg Sc}, t\rangle)) &= e^{-\lambda\Delta}
\end{aligned}$$

where $e^{-\lambda\Delta} = 0.5$ when Δ is one day.

The other terms in Formula 7.9. N2. N3. N4. N5. and N6. we will assume to be. respectively. 1. 0. 0. 1. and 0. When the mechanic on duty cleans up the shop. he is supposed to put all of the tools in their appropriate places. In particular. Wrench14 and Screwdriver31 are supposed to be returned to Room101. We assume that the mechanic is very diligent:

$$\begin{aligned} \Pr(\langle E_{Wr}, t+\epsilon\rangle | \langle Cl. t\rangle) &= 1.0 \\ \Pr(\langle E_{Sc}, t+\epsilon\rangle | \langle Cl. t\rangle) &= 1.0 \end{aligned}$$

Fred's competence in assembling widgets depends upon his tools being in the right place. In particular. if Screwdriver31 and Wrench14 are in Room101. then it is certain that Fred will successfully assemble Widget17.

$$\Pr(\langle E_{Wi}, t+\epsilon\rangle | \langle Wr. t\rangle \wedge \langle Sc. t\rangle \wedge \langle As, t\rangle) = 1.0$$

Let *T0* correspond to 12:00 PM 2/29/88. and *T1* correspond to 12:00 PM on the following day. Assume that ϵ is negligible given the events we are concerned with (*i.e.*. we will add or subtract ϵ in order to simplify the analysis).

$$\begin{aligned} \Pr(\langle Cl. T0\rangle) &= 0.7 \\ \Pr(\langle As. T1\rangle) &= 1.0 \end{aligned}$$

We are interested in assigning the propositions of the form $\langle \varphi, t\rangle$ a certainty measure consistent with the axioms of probability theory. We will work through an example showing how one might derive such a measure, noting some of the assumptions required to make the derivations follow from the problem spetification and the axioms of probability. In the following. we will denote this measure of belief by Bel. What can we say about $\text{Bel}(\langle Wi, T1+\epsilon\rangle)$? In this particular example. we begin with

$$\begin{aligned} &\text{Bel}(\langle Wi, T1+\epsilon\rangle) \\ &= \Pr(\langle E_{Wi}. T1+\epsilon\rangle) \\ &= \Pr(\langle E_{Wi}, T1+\epsilon\rangle | \langle Wr. T1\rangle \wedge \langle Sc. T1\rangle \wedge \langle As. T1\rangle) \\ &\quad * \Pr(\langle Wr. T1\rangle \wedge \langle Sc. T1\rangle \wedge \langle As. T1\rangle) \\ &= \Pr(\langle Wr. T1\rangle \wedge \langle Sc. T1\rangle \wedge \langle As. T1\rangle) \\ &= \Pr(\langle Wr. T1\rangle \wedge \langle Sc. T1\rangle) \end{aligned}$$

The first step follows from our interpretation of E_{Wr}, and the fact that there is no additional evidence for or against Wr at $T1+\epsilon$. The second step employs the addition rule and the assumption that the assembly will fail to have the effect of $\langle E_{Wr}, T1\rangle$ if any one of $\langle Wr, T1\rangle$, $\langle Sc, T1\rangle$, or $\langle As, T1\rangle$ is false. The third step relies on the fact that assembly is always successful given that the attempt is made and Wrench14 and Screwdriver31 are in Room101. The last step depends on the assumption that the evidence supporting $\langle Wr \wedge Sc, T1\rangle$ and $\langle As, T1\rangle$ are independent. The assumption is warranted in this case given that the particular instance of As occurring at $T1$ does not affect $Wr \wedge Sc$ at $T1$, and the evidence for As at $T1$ is independent of any events prior to $T1$. Note that, if the evidence for As at $T1$ involved events prior to $T1$, then the analysis would be more involved. It is clear that $\Pr(\langle Wr, T1\rangle) \geq 0.35$, and that $\Pr(\langle Sc, T1\rangle) \geq 0.35$; unfortunately, we cannot simply combine this information to obtain an estimate of $\Pr(\langle Wr \wedge Sc, T1\rangle)$, since the evidence supporting these two claims is dependent. We can, however, determine that

$$\begin{aligned}
&\Pr(\langle Wr, T1\rangle \wedge \langle Sc, T1\rangle)\\
&\quad = \Pr(\langle Wr, T1\rangle \wedge \langle Sc, T1\rangle | \langle Wr, T0\rangle \wedge \langle Sc, T0\rangle) \Pr(\langle Wr, T0\rangle \wedge \langle Sc, T0\rangle)\\
&\quad = \Pr(\langle Wr, T1\rangle | \langle Wr, T0\rangle \wedge \langle Sc, T0\rangle)\\
&\qquad * \Pr(\langle Sc, T1\rangle | \langle Wr, T0\rangle \wedge \langle Sc, T0\rangle) \Pr(\langle Wr, T0\rangle \wedge \langle Sc, T0\rangle)\\
&\quad = \Pr(\langle Wr, T0\rangle \wedge \langle Sc, T0\rangle) * 0.5 * 0.5\\
&\quad = \Pr(\langle E_{Wr}, T0\rangle \wedge \langle E_{Sc}, T0\rangle) * 0.5 * 0.5\\
&\quad = \Pr(\langle E_{Wr}, T0+\epsilon\rangle \wedge \langle E_{Sc}, T0+\epsilon\rangle | \langle Cl, T0\rangle) \Pr(\langle Cl, T0\rangle) * 0.5 * 0.5\\
&\quad = 0.7 * 0.5 * 0.5\\
&\quad = 0.175
\end{aligned}$$

assuming that there is no evidence concerning events that are known to affect either Wr or Sc in the interval from $T0$ to $T1$, that Wr and Sc are independent, and that E_{Wr} and E_{Sc} are conditionally independent of one another given Cl.

Throughout our analysis, we were forced to make assumptions of independence. In many cases, such assumptions are unwarranted or introduce inconsistencies. The inference process is further complicated by the fact that probabilistic constraints tend to propagate both forward and backward in time. This bi-directional flow of evidence can render the analysis described above useless. In the next section, we consider a model that simplifies specifying independence assumptions, and that allows us to handle both forward and backward propagation of probabilistic constraints.

7.4.4 A Model for Reasoning About Change

In this section, we take a slight modification of Formula 7.9 as the basis for a model of persistence. Formula 7.9 predicts $\langle P, t\rangle$ on the basis of $\langle P, t - \Delta\rangle$, $\langle E_P, t\rangle$, and $\langle E_{\neg P}, t\rangle$, where Δ is allowed to vary. In the model presented in this section, we only consider pairs of consecutive time points, t and $t + \delta$, and arrange things so that the value of a fluent at time t is completely determined by the state of the world at δ in the past. In Formula 7.9, we interpret events of type E_P occurring at t as providing evidence for P being true at t. In our new model, we interpret events of type E_P occurring at t as providing evidence for P being true at $t + \delta$. This reinterpretation is not strictly necessary, but we prefer it since the expressiveness of the resulting models can easily be characterized in terms of the properties of Markov processes. In our new model, we predict $A \equiv \langle P, t + \delta\rangle$ by conditioning on

$$\begin{aligned} C_1 &\equiv \langle P, t\rangle \\ C_2 &\equiv \langle E_P, t\rangle \\ C_3 &\equiv \langle E_{\neg P}, t\rangle \end{aligned}$$

and specify a complete model for the persistence of P as

$$\Pr(A) = \sum \Pr(A | C_1 \wedge C_2 \wedge C_3)\ \Pr(C_1 \wedge C_2 \wedge C_3)$$

where the sum is over the eight possible truth assignments for the variables C_1, C_2, and C_3. Note that this model requires that we have probabilities of the form $\Pr(A|C_1 \wedge C_2 \wedge C_3)$ and $\Pr(C_1 \wedge C_2 \wedge C_3)$ for all possible valuations of the C_i.

In the following, we will make use of a network model that will serve to clearly indicate the necessary indepedence assumptions. We will use the generic term belief network to refer to a network that satisfies the following basic properties common to all three of the above representations. A belief network represents the variables or propositions of a probabilistic theory as nodes in a graph. The variables in our networks correspond to propositional variables of the form $\langle \varphi, t\rangle$. Dependence between two variables is indicated by a directed arc between the two nodes associated with the variables.

Because dependence is always indicated by an arc, belief networks make it easy to identify the conditional independence inherent in a model simply by inspecting the graph. Two nodes which are linked via a common

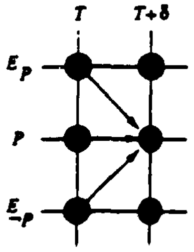


Figure 7.18: The evidence for P at time $T+\delta$

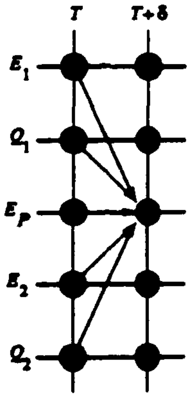


Figure 7.19: The evidence for E_P at time $T+\delta$

neighbor, but for which there are no other connecting paths are conditionally independent given the common node. For instance, in the models described in this section, $\langle P, t-\delta\rangle$ is independent of $\langle P, t+\delta\rangle$ given $\langle P, t\rangle$. Belief networks make it easy to construct and verify the correctness and reasonableness of a model directly in terms of the corresponding graphical representation. Our model for persistence can be represented by the network shown in Figure 7.18. As soon as we provide a model for causation, we will show how this simple model for persistence can be embedded in a more complex model for reasoning about change over time.

Generally, we expect that the cause-and-effect relations involving E_P will be specified in terms of constraints of the form:

$$\begin{aligned}\Pr(\langle E_P, t+\delta\rangle | \langle E_1, t\rangle \wedge \langle Q_1, t\rangle) &= \pi_1\\ \Pr(\langle E_P, t+\delta\rangle | \langle E_2, t\rangle \wedge \langle Q_2, t\rangle) &= \pi_2\end{aligned}$$

$$\ldots$$

$$\Pr(\langle E_P, t+\delta\rangle | \langle E_n, t\rangle \wedge \langle Q_n, t\rangle) \;=\; \pi_n$$

However, to specify a complete model, we will need some more information. To predict $A \equiv \langle E_P, t+\delta\rangle$, we condition on

$$\begin{aligned} C_1 &\equiv \langle E_1, t\rangle \wedge \langle Q_1, t\rangle \\ C_2 &\equiv \langle E_2, t\rangle \wedge \langle Q_2, t\rangle \\ &\ldots \\ C_n &\equiv \langle E_n, t\rangle \wedge \langle Q_n, t\rangle \end{aligned}$$

and specify a complete model as:

$$\Pr(A) = \sum \Pr(A|C_1 \wedge C_2 \wedge \ldots \wedge C_n)\ \Pr(C_1 \wedge C_2 \wedge \ldots \wedge C_n)$$

Note that we need on the order of 2^n probabilities corresponding to the 2^n possible valuations of the propositional variables C_1 through C_n to specify this model. The associated belief network is shown in Figure 7.19. Similar networks would be constructed for event types other than those involving propositions becoming true or false.

Now we can construct a complete model for reasoning about change over time. Figure 7.20 illustrates the temporal belief network for such a complete model. For each propositional variable of the form $\langle \varphi, t\rangle$, there is a node in the belief network. The arcs are specified according to the isolated models for persistence and causation illustrated in Figure 7.18 and Figure 7.19. Following Pearl (1988), we can write down the unique distribution corresponding to the model shown in Figure 7.20 as

$$\Pr(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} \Pr(x_i | S_i) \Pr(S_i)$$

where the x_i denote the propositional variables in the model, and S_i is the conjunction of the propositional variables associated with those nodes for which there exist arcs to x_i in the network.

As a specific instance of a temporal belief network, we reconsider the factory example of Section 7.4.3. We will need models for the persistence of wrenches and screwdrivers remaining in place, and models for reasoning about the consequences of cleaning and assembling actions. Figure 7.21.i shows a portion of a belief network dedicated to modeling the persistence of Wr (*i.e.*, the proposition corresponding to Wrench14 being in Room101).

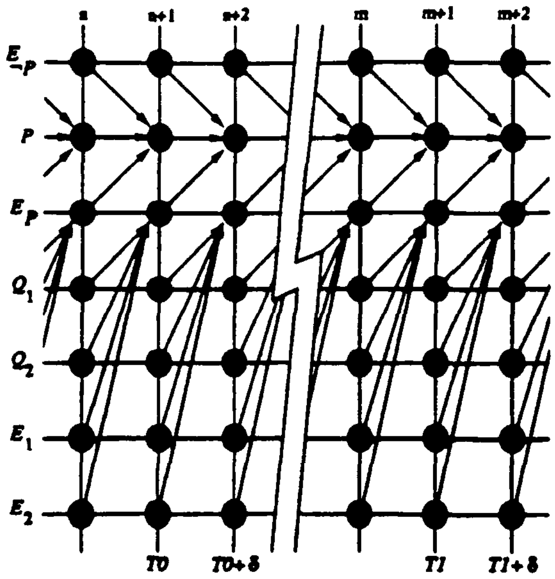


Figure 7.20: A temporal belief network

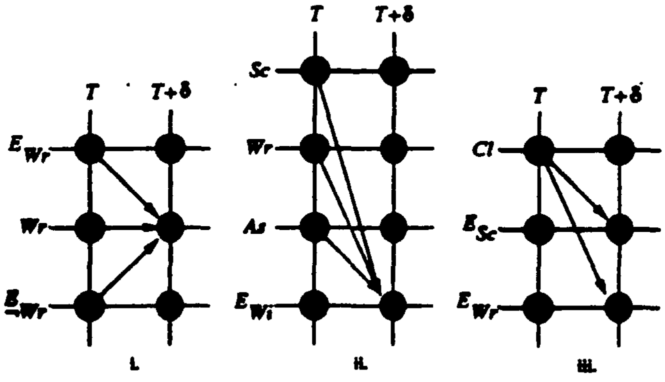


Figure 7.21: Models for the factory example

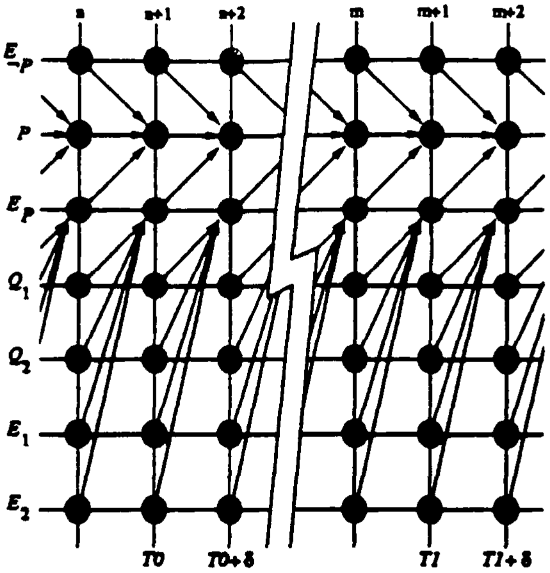


Figure 7.20: A temporal belief network

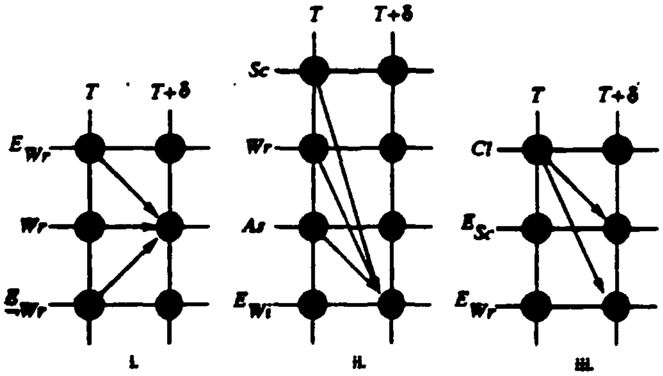


Figure 7.21: Models for the factory example

In order to completely specify the model for Wr persisting. we need the following information:

$\Pr(\langle Wr, t\rangle \mid \ldots)$	$\langle Wr, t - \Delta\rangle$	$\langle E_{Wr}, t - \Delta\rangle$	$\langle E_{\neg Wr}, t - \Delta\rangle$
$\epsilon^{-\lambda\Delta}$	True	False	False
$\epsilon^{-\lambda\Delta}$	True	True	False
0.0	True	False	True
0.0	False	False	False
$e^{-\lambda\Delta}$	False	True	False
0.0	False	False	True
—	True	True	True
—	False	True	True

The first six entries entries in the table correspond to terms N1–6 in Formula 7.9. Note that the entries corresponding to N2 and N5—assumed to be 1 in Section 7.4.3—are now the same as N1 to account for our revised interpretation of events of type E_P.

Figure 7.21.ii shows a portion of a belief net for modeling the effects of the assembly action. The complete model is specified as follows:

$\Pr(\langle E_{Wr}, t\rangle \mid \ldots)$	$\langle Sc, t - \epsilon\rangle$	$\langle Wr, t - \epsilon\rangle$	$\langle As, t - \epsilon\rangle$
0.0	False	False	False
0.0	True	False	False
0.0	False	True	False
0.0	True	True	False
0.0	False	False	True
0.0	True	False	True
0.0	False	True	True
1.0	True	True	True

Finally, Figure 7.21.iii shows a portion of a belief net for modeling the effects of the cleaning action. The complete model for the effect of cleaning on the location of Wrench14 are shown below:

$\Pr(\langle E_{Wr}, t\rangle \mid \ldots)$	$\langle Cl, t - \epsilon\rangle$
0.0	False
1.0	True

and similarly for the effect of cleaning on the location of Screwdriver31:

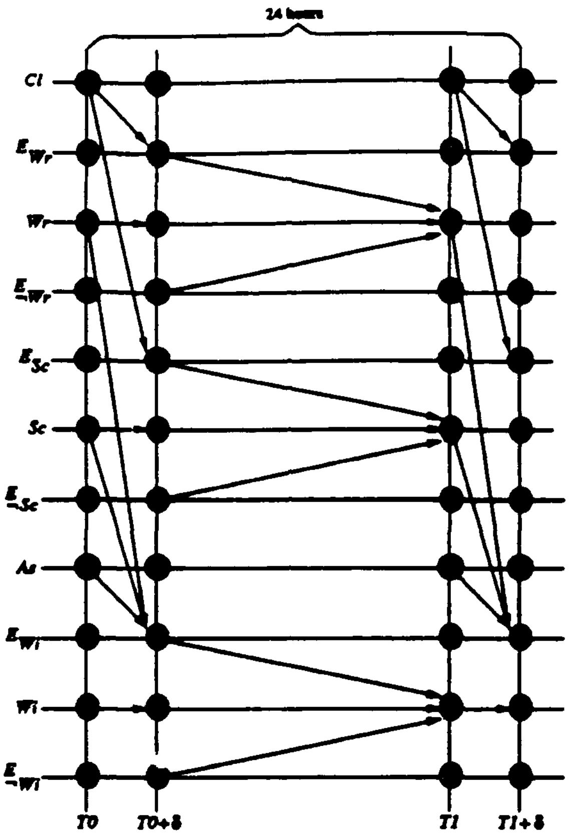


Figure 7.22: A belief network for the factory example

$\Pr(\langle E_{Sc}, t\rangle \mid \ldots)$	$\langle Cl, t-\epsilon\rangle$
0.0	False
1.0	True

In the discussion of the general model, the amount of time separating time points was assumed to be the same for all pairs of consecutive time points. In reasoning about the factory example, it will be useful to have the time separating pairs of consecutive time points differ, and to have different models for handling different separations. We will need time points close together for propagating the (almost immediate) consequences of actions, and time points separated by several hours so as not to incur the computational expense of reasoning about intervals of time during which little of interest happens. To reduce the complexity of the network for the factory example. we assume that evidence concerning the occurrence of actions such as cleaning and assembling is always with regard to the end points of 24 hour intervals. Figure 7.22 shows the complete network for the factory example. Note that. since the evidence for actions appears only at 24 hour intervals. we encode the models for action only at the time points $T0$ and $T1$: similarly. since additional evidence for events of type E_P is only available at $T0 + \epsilon$ and $T1 + \epsilon$, we use a simpler model for persistence at $T0$ and $T1$ in which. for example. $\langle Wr. T0 + \epsilon \rangle$ is completely determined by $\langle Wr. T0 \rangle$. If we assume a prior probability of 0 for all nodes without predecessors in Figure 7.22 excepting $\langle Cl. T0 \rangle$ and $\langle As. T1 \rangle$ which are, respectively, 0.7 and 1.0, then $\Pr(\langle Ew_i, T1 + \epsilon \rangle)$ is 0.175 in the unique posterior distribution determined by the network. This is the same as that established by the analysis of Section 7.4.3. but, in this case, we have made all of our assumptions of independence explicit in the structure of the temporal belief network.

It is straightforward to extend the model described above to account for new observations and updating beliefs. Suppose we have the observations $o_1, o_2, \ldots, o_n$, where each observation is of the form $\langle O, t \rangle$ and O is an event type corresponding to a particular type of observation. We assume some prior distribution specified in terms of constraints of the form:

$$\Pr(\langle O, t \rangle) = 0.001$$

There are also constraints indicating prior belief regarding the occurrence of events other than observations. For instance, we might have

$$\Pr(\langle E, t \rangle) = 0.001.$$

Observations are related to events by constraints such as

$$\Pr(\langle E. t \rangle | \langle O. t \rangle) = 0.70$$

and

$$\Pr(\langle E. t \rangle | \neg \langle O. t \rangle) = 0.025.$$

To update an agent's beliefs you can either change the priors:

$$\Pr(\langle O. t \rangle) = 1.0$$

or you can compute the posterior distribution:

$$\mathrm{Bel}(A) = \Pr(A | o_1, o_2, \ldots, o_n).$$

Most of the standard techniques for representing and reasoning about evidence in belief networks apply directly to our model.

Need material on the expressive limitations of this model. Relation to Markov processes and Markov chains.

Suppose that the instantaneous state of the world can be completely specified in terms of a vector of values assigned to a finite set of boolean variables $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, and suppose further that the environment can be accurately modeled as a Markov process in which time is discrete and the state space Ω corresponds to all possible valuations of the variables in $\mathcal{P}$. Given such a model including a transition matrix defined on Ω, we can generate a temporal belief network to compute the probability of any proposition in $\mathcal{P}$ being true at any time t based upon evidence concerning the values of variables in $\mathcal{P}$ at various times, and do so in accord with the transition probabilities specified in the Markov model. Conversely, given a temporal belief network such that, for all t and $P \in \mathcal{P}$, all of the predecessors of $\langle P, t\rangle$ are in the set $\{\langle P_i, t - \delta\rangle\}$, the network is said to satisfy the Markov property for temporal belief networks, and, from this network, one can construct an equivalent Markov chain.

The reason that one might use a fluent-and-event-based temporal belief network model rather than an equivalent state-based Markov model is because the belief network representation facilitates reasoning of the sort required for applications in planning and decision support (*e.g.*, computing answers to questions of the form, "What is the probability of P at t given everything else we know about the situation?"). These same answers can be computed using the Markov model, but the process is considerably less direct.

Satisfying the Markov property for temporal belief networks allows us to establish the connection between temporal belief networks and Markov chains, but it sometimes results in unintuitive network structures. Introducing a delay between an action and its consequences may appear reasonable give the intuition that causes precede effects. However, introducing a delay between E_P and P simply to ensure the Markov property may seem a little extreme. We can eliminate the delay between E_P and P by returning to the model for persistence in Formula 7.9. The resulting networks do not satisfy the Markov property described in this section, but they are perfectly legitimate temporal belief nets and provide a somewhat more intuitive model for representing change than networks that do satisfy the Markov property.

7.4.5 Fundamental Problems in Temporal Reasoning

Given that our model addresses many of the same problems that concern logicians working on temporal logic, we will briefly mention how our model deals with certain classic problems in temporal reasoning: the frame, ramification, and qualification problems. We will begin by considering the frame problem stated in probabilistic terms: "Does our model accurately capture our expectations regarding fluents that are considered *not* likely to change as a consequence of a particular event occurring?" The answer is yes insofar as frame axioms can be said to solve the frame problem in temporal logic: persistence constraints are the probabilistic equivalent of frame axioms.

In considering the ramification problem, we will consider two possible interpretations. First, "Does our model enable us to compute appropriate expectations regarding the value of a particular fluent at a particular point in time without bothering with a myriad of seemingly unimportant consequences?" The answer to this is a resounding no: our model commits us to predicting every possible consequence of every possible action no matter how implausible. A second interpretation (or perhaps facet is a better word) of the ramification problem is "Does our model enable us to handle additional consequences that follow from a set of causal predictions?" For instance, if A is in box B and I move B to a new location, I should be able to predict that A will be in the new location along with B. Our model provides no provision at all for this sort of reasoning. The basic idea of Bayesian inference can be extended to handle this sort of reasoning, but we have not investigated this to date.

The last problem we consider concerns reasoning about exceptions involving the rules governing cause-and-effect relationships. Does our model solve the qualification problem? That is to say, "Does our model accurately capture our expectations regarding the possible exceptions to knowledge about cause-and-effect relationships?" The answer is yes: conditional probabilities would seem to be exactly suited for this sort of reasoning. It should be noted, however, that our model imposes a considerable burden on the person setting up the model. The model described in this section requires specifying all possible causes for each possible effect and the probability of each effect for every possible combination of possible causes. It is not clear, however, that one can get away with less. Given the problems inherent in eliciting such information from experts, it would appear that we will have to automate the process of setting up our probabilistic models.

The third example is drawn from [12] and concerns the sequential de-

cision problem for the mobile target localization (MTL) problem. Be sure to address the issue concerning the duration of the time interval separating points in the temporal Bayes network. There are two possible approaches for the MTL problem. Either the intervals are of a fixed duration independent of the action performed, or they are dependent on the action performed in which case additional arcs have to be added between the action nodes for the robot at one point in time and all of the other nodes at the next point in time. In the first approach, the model is simple and control is tricky; in the second approach, the model is complex and control is simple.

7.5 Sequential Decision Making

In this section, we consider an approach to building planning and control systems that integrates sensor fusion, prediction, and sequential decision making. The approach is based on Bayesian decision theory, and involves encoding the underlying planning and control problem in terms of probabilistic models We illustrate the approach using a robotics problem that requires spatial and temporal reasoning under uncertainty and time pressure. We use the estimated computational cost of evaluation to justify representational tradeoffs required for practical application.

In this secton, we view planning in terms of enumerating a set of possible courses of action, evaluating the consequences of those courses of action, and selecting a course of action whose consequences maximize a particular performance (or *value*) function. We adopt Bayesian decision theory as the theoretical framework for our discussion, since it provides a convenient basis for dealing with decision making under uncertainty.

One interesting thing about most planning problems is that the results of actions can increase our knowledge, potentially improving our ability to make decisions. From a decision theoretic perspective, there is no difference between actions that involve sensing or movement to facilitate sensing and any other actions; a decision maker simply tries to choose actions that maximize expected value. In the approach described in this section, an agent engaged in a particular perceptual task selects a set of sensor views by physically moving about.

Having committed to a decision theoretic approach, there are specific problems that we have to deal with. The most difficult concern representing the problem and obtaining the necessary statistics to quantify the underlying decision model. In the robotics problems we are working on, the latter is relatively straightforward, and so we will concern ourselves primarily with the former.

In building a decision model for control purposes, it is not enough to write down all of your preferences and expectations; this information might provide the basis for constructing some decision model, but it will likely be impractical from a computational standpoint. It is frustrating when you know what you want to compute but cannot afford the time to do so. Some researchers respond by saying that eventually computing machinery will be up to the task and ignore the computational difficulties. It is our contention, however, that the combinatorics inherent in sequential decision making will continue to outstrip computing technologies.

In the following, we describe a concrete problem to ground our discussion, present the general sequential decision making model and its application to the concrete problem, show how to estimate the computational costs associated with using the model, and, finally, describe how to reduce those costs to manageable levels by making various representational tradeoffs.

7.5.1 Mobile Target Localization

The application that we have chosen to illustrate our approach involves a mobile robot navigating and tracking moving targets in a cluttered environment. The robot is provided with sonar and rudimentary vision. The moving target could be a person or another mobile robot. The mobile base consists of a holonomic (turn-in-place) synchro-drive robot equipped with a CCD camera mounted on a pan-and-tilt head, and 8 fixed Polaroid sonar sensors arranged in pairs directed forward, backward, right, and left.

The robot's task is to detect and track moving objects, reporting their location in the coordinate system of a global map. The environment consists of one floor of an office building. The robot is supplied with a floor plan of the office showing the position of permanent walls and major pieces of furniture such as desks and tables. Smaller pieces of furniture, potted plants and other assorted clutter constitute obstacles that the robot has to detect and avoid.

We assume that there is error in the robot's movement requiring it to continually estimate its position with respect to the floor plan so as not to become lost. Position estimation (*localization*) is performed by having the robot track *beacons* corresponding to walls and corners and then use these beacons to reduce error in its position estimate.

Localization and tracking are frequently at odds with one another. A particular localization strategy may reduce position errors while making tracking difficult, or improve tracking while losing registration with the global map. The trick is to balance the demands of localization against the demands of tracking. The mobile target localization (MTL) problem is particularly appropriate for planning research as it requires considerable complexity in terms of temporal and spatial representation, and involves time pressure and uncertainty in sensing and action.

7.5.2 Model for Time and Action

In this section, we provide a decision model for the MTL problem. To specify the model, we quantize the space in which the robot and its target are embedded. A natural quantization can be derived from the robot's sensory capabilities.

The robot's sonar sensors enable it to recognize particular patterns of free space corresponding to various configurations of walls and other permanent objects in its environment (e.g., corridors, L junctions and T junctions). We tessellate the area of the global map into regions such that the same pattern is detectable anywhere within a given region. This tessellation provides a set of locations $\mathcal{L}$ corresponding to the regions that are used to encode the location of both the robot and its target.

Our decision model includes two variables S_T and S_R, where S_T represents the location of the target and ranges over $\mathcal{L}$, and S_R represents the location and orientation of the robot and ranges over an extension of $\mathcal{L}$ including orientation information specific to each type of location. For any particular instance of the MTL problem, we assume that a geometric description of the environment is provided in the form of a CAD model. Given this geometric description and a model for the robot's sensors, we generate $\mathcal{L}$, S_R, and S_T.

The model described here is based on the approach of Section 7.4. Given a set of discrete variables, $\mathcal{X}$, and a finite ordered set of time points, $\mathcal{T}$, we construct a set of chance nodes, $\mathcal{C} = \mathcal{X} \times \mathcal{T}$, where each element of $\mathcal{C}$ corresponds to the value of some particular $x \in \mathcal{X}$ at some $t \in \mathcal{T}$. Let $\mathcal{C}_t$ correspond to the subset of $\mathcal{C}$ restricted to t. The temporal belief networks discussed in this section are distinguished by the following Markov property:

$$\Pr(\mathcal{C}_t|\mathcal{C}_{t-1}, \mathcal{C}_{t-2}, \ldots) = \Pr(\mathcal{C}_t|\mathcal{C}_{t-1}).$$

Let S_R and S_T be variables ranging over the possible locations of the robot and the target respectively. Let A_R be a variable ranging over the actext thesis/graph smaller

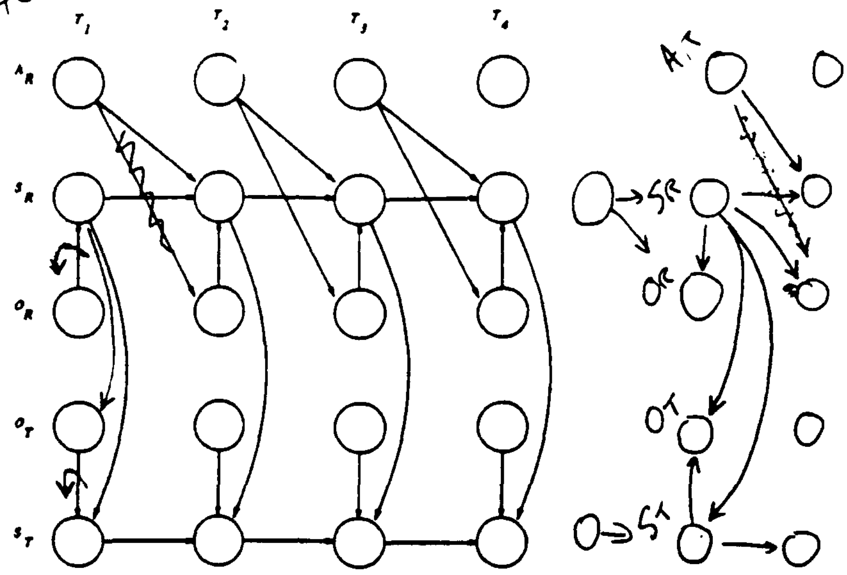


Figure 7.23: Probabilistic model for the MTL problem

tions available to the robot. At any given point in time, the robot can make observations regarding its position with respect to nearby walls and corners and the target's position with respect to the robot. Let O_R and O_T be variables ranging these observations with respect to the robot's surroundings and the target's relative location.

Figure 7.23 shows a temporal belief network for $\mathcal{X} = \{S_R, S_T, A_R, O_R, O_T\}$ and $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$. To quantify the model shown in Figure 7.23, we have to provide distributions for each of the variables in $\mathcal{X} \times \mathcal{T}$. We assume that the model does not depend on time, and, hence,.we need only provide one probability distribution for each $x \in \mathcal{X}$. For instance, the conditional probability distribution for S_T,

$$\Pr(\langle S_T, t\rangle | \langle S_T, t{-}1\rangle, \langle O_T, t\rangle, \langle S_R, t\rangle),$$

is the same for any $t \in \mathcal{T}$. The numbers for the probability distributions can be obtained by experimentation without regard to any particular global map.

In a practical model consisting of more than just the four time points shown in Figure 7.23, some points will refer to the past and some to the future. One particular point is designated the current time or *Now*. Representing the past and present will allow us to incorporate evidence into the

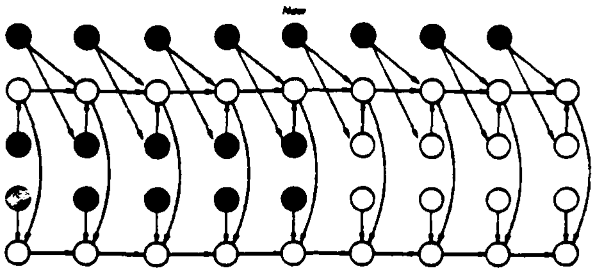


Figure 7.24: Evidence and action sequences

model. By convention, the nodes corresponding to observations are meant to indicate observations *completed* at the associated time point, and nodes corresponding to actions are meant to indicate actions *initiated* at the associated time point. The actions of the robot at past time points and the observations of the robot at past and present time points serve as evidence to provide conditioning events for computing a posterior distribution. For instance, having observed σ at T, denoted $\langle O_R{=}\sigma, T\rangle$, and initiated α at T–1, denoted $\langle A_R{=}\alpha, T{-}1\rangle$, we will want to compute the posterior distribution for S_R at T given the evidence:

$$\Pr(\langle S_R{=}\omega, T\rangle, \omega \in \Omega_{S_R} | \langle O_R{=}\sigma, T\rangle, \langle A_R{=}\alpha, T{-}1\rangle).$$

To update the model as time passes, all of the evidence nodes are shifted into the past, discarding the oldest evidence in the process. Figure 7.24 shows a network with nine time points. The lighter shaded nodes correspond to evidence. As new actions are initiated and observations are made, the appropriate nodes are instantiated as conditioning nodes, and all of the evidence is shifted to the left by one time point.

The darker shaded nodes shown in Figure 7.24 indicate nodes that are instantiated in the process of evaluating possible sequences of actions. For evaluation purposes, we employ a simple *time-separable* value function. By time separable, we mean that the total value is a (perhaps weighted) sum of the value at the different time points. If V_t is the value function at time t, then the total value, V, is defined as

$$V = \sum_{t \in T} \gamma(t) V_t,$$

where $\gamma : T \rightarrow \{x | 0 \leq x \leq 1\}$ is a decreasing function of time used to discount the impact of future consequences. Since our model assumes a finite T. we already discount some future consequences by ignoring them altogether; γ just gives us a little more control over the immediate future. For V_t. we use the following function

$$V_t = -\sum_{\omega_i,\omega_j \in \Omega_{S_T}} \Pr(\langle S_T{=}\omega_i.t\rangle)\Pr(\langle S_T{=}\omega_j.t\rangle)\mathrm{Dist}(\omega_i.\omega_j).$$

where Dist : $\Omega_{S_T} \times \Omega_{S_T} \rightarrow \Re$ determines the relative Euclidean distance between pairs of locations. The V_t function reflects how much uncertainty there is in the expected location for the target. For instance, if the distribution for $\langle S_T, t\rangle$ is strongly weighted toward one possible location in Ω_{S_T}, then V_t will be close to zero. The more places the target could be and the further their relative distance, the more negative V_t.

The actions in Ω_{A_R} consist of tracking and localization routines (e.g., move along the wall on your left until you reach a corner). Each action has its own termination criteria (e.g., reaching a corner). We assume that the robot has a set of strategies, $\mathcal{S}$, consisting of sequences of such actions, where the length of sequences in $\mathcal{S}$ is limited by the number of present and future time points. For the network shown in Figure 7.24, we have

$$\mathcal{S} \subset \Omega_{A_R} \times \Omega_{A_R} \times \Omega_{A_R} \times \Omega_{A_R}.$$

The size of $\mathcal{S}$ is rather important. since we propose to evaluate the network $|\mathcal{S}|$ times at every decision point. The strategy with the highest expected value is that strategy, $\varphi = \alpha_0.\alpha_1,\alpha_2,\alpha_3$. for which V is a maximum. conditioning on $\langle A_r{=}\alpha_0, Now\rangle$. $\langle A_r{=}\alpha_1, Now{+}1\rangle$. $\langle A_r{=}\alpha_2, Now{+}2\rangle$. and $\langle A_r{=}\alpha_3, Now{+}3\rangle$. The best strategy to pursue is reevaluated every time that an action terminates.

We use Jensen's [21] variation on Lauritzen and Spiegelhalter's [25] algorithm to evaluate the decision network. Jensen's algorithm involves constructing a hyper graph (called a *clique tree*) whose vertices correspond to the (maximal) cliques of the chordal graph formed by triangulating the undirected graph obtained by first connecting the parents of each node in the network and then eliminating the directions on all of the edges. The cost of evaluating a Bayesian network using this algorithm is largely determined by the sizes of the state spaces formed by taking the cross product of the state spaces of the nodes in each vertex (clique) of the clique tree.

Following Kanazawa [22], we can obtain an accurate estimate of the cost of evaluating a Bayesian network, $G = (V, E)$, using Jensen's algorithm. Let $C = \{C_i\}$ be the set of (maximal) cliques in the chordal graph described in the previous paragraph, where each clique represents a subset of V. We define the function, card : $C \rightarrow \{1, \ldots, |C| - 1\}$, so that card($C_i$) is the rank of the highest ranked node in C_i, where rank is determined by the maximal cardinality ordering of V (see [32]). We define the function, adj : $C \rightarrow 2^C$, by:

$$\text{adj}(C_i) = \{C_j | (C_j \neq C_i) \wedge (C_i \cap C_j \neq \emptyset)\}.$$

The clique tree for G is constructed as follows. Each clique $C_i \in C$ is connected to the clique C_j in adj(C_i) that has lower rank by card(.) and has the highest number of nodes in common with C_i (ties are broken arbitrarily). Whenever we connect two cliques C_i and C_j, we create the *separation set* $S_{ij} = C_i \cap C_j$. The set of separation sets S is all the S_{ij}'s. We define the function, sep : $C \rightarrow 2^S$, by:

$$\text{sep}(C_i) = \{S_{jk} | S_{jk} \in S, (j = i) \vee (k = i)\}.$$

Finally, we define the *weight* of C_i, $w_i = \prod_{n \in C_i} |\Omega_n|$, where Ω_n is the state space of node n. The cost of computation is proportional to $\sum_{C_i \in C} w_i |\text{sep}(C_i)|$. We refer to this cost estimate as the *clique-tree cost*.

The approach described in this section allows us to integrate prediction, observation, and control in a single model. It also allows us to handle uncertainty in sensing, movement, and modeling. Behavioral properties emerge as a consequence of the probabilistic model and the value function provided, not as a consequence of explicitly programming specific behaviors. The main drawback of the approach is that, while the model is quite compact, the computational costs involved in evaluating the model can easily get out of hand. For instance, in our model for the MTL problem, the clique-tree cost is bounded from below by the product of $|T|$, $|\Omega_{S_T}|^2$, and $|\Omega_{S_R}|^2$. In the next section, we provide several methods that, taken together, allow us to reduce computational costs to practical levels.

7.5.3 Coping with Complexity

To reduce the cost of evaluating the MTL decision model, we use the following three methods: (i) carefully tailor the spatial representation to the robot's sensory capabilities, reducing the size of the state space for the spatial variables in the decision model, (ii) enable the robot to dynamically narrow the range of the spatial variables using heuristics to further reduce the size of the state space for the spatial variables, and (iii) consider only a few candidate action sequences from a fixed library of tracking strategies

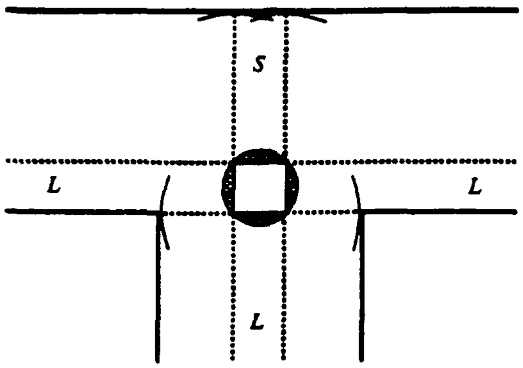


Figure 7.25: Sonar data entering a T junction

by taking into account the reduced state space of the spatial variables. In the rest of this section, we consider each of these three methods.

The use of a high-resolution representation of space has disadvantages in the model proposed here: increasing the resolution of the representation of space results in an increase in the sizes of Ω_{S_R} and Ω_{S_T}, and thus raises the cost of evaluating the network. Keeping the sizes of Ω_{S_R} and Ω_{S_T} small makes the task of evaluating the model we propose feasible.

A further consideration arises from the real-world sensory and data processing systems available to our robot. Finer-resolution representations of space place larger demands on the robot's on-board system in terms of both run-time processing time and sensor accuracy. To allow our robot to achieve (near) real-time performance, it seems appropriate to limit the representation to that level of detail that can be obtained economically from the hardware available.

In our current implementation, we have 8 sonar transducers positioned on a square platform, two to a side, spaced about 25 cm. apart. We take distance readings from each transducer, and threshold the values at about 1 meter. Anything above the threshold is "long," anything below is "short." The readings along each side are then combined by voting, with ties going to "long." In this way, the data from the sonar is reduced to 4 bits. Figure 7.25 shows the result of this scheme on entering a T junction. In addition, we use the shaft encoders on our platform to provide very rough metric information for the decision model. Currently, 2 additional bits are used for this purpose, but only when the robot is positioned in a hallway, which corresponds to only one sonar configuration. So the total number of possible states for O_R

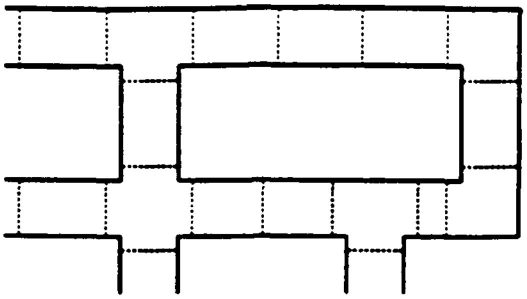


Figure 7.26: Tessellation of office layout

is 19. 15 for various kinds of hallway junctions and 4 more for corridors.

This technique results in a tessellation of space like that shown in Figure 7.26. Our experiments have shown that this tessellation is quite robust in the sense that the readings are consistent anywhere in a given tile. The exception to this occurs when the robot is not well-aligned with the surrounding walls. In these cases, reflections frequently make the data unreliable. One of the tasks of the controllers that underlie the actions described in the previous sections is to maintain good alignment, or achieve it if it is lost.

In addition to reducing the size of the overall spatial representation, we can restrict the range of particular spatial variables on the basis of evidence not explicitly accounted for in the decision model (e.g., odometry and compass information). For instance, if we know that the robot is in one of two locations at time 1 and the robot can move at most a single location during a given time step, then $\langle S_R, 1\rangle$ ranges over the two locations, and, for $i > 1$, $\langle S_R, i\rangle$ need only range over the locations in or adjacent to those in $\langle S_R, i{-}1\rangle$. Similar restrictions can be obtained for S_T. For models with limited lookahead (i.e., small $|T|$), these restrictions can result in significant computational savings.

Consider a temporal Bayesian network of the form shown in Figure 7.23 with n steps of lookahead. Let $\langle X, i\rangle$ represent an element of $\{S_R, S_T, A_R, O_R, O_T\} \times \{1, \ldots, n\}$. The largest cliques in one possible[3] clique tree for this network consist of sets of variables of the form:

$$\{\langle S_R, i\rangle, \langle S_R, i{+}1\rangle, \langle S_T, i\rangle, \langle S_T, i{+}1\rangle\}$$

[3] The triangulation algorithm attempts to minimize the size of the largest clique in the resulting chordal graph. There may be more than one way to triangulate a graph so as to minimize the clique size.

State space size	Number of time points 3	 5	 8
Constant (6)	40914 (0.58)	78066 (1.11)	133794 (1.90)
Constant (16)	624944 (8.87)	1232176 (17.49)	2143024 (30.42)
Constant (30)	3846330 (54.60)	7669530 (108.86)	13404330 (190.26)
Linear ($2t+1$)	5844 (0.08)	55088 (0.78)	433759 (6.16)
Quadratic (t^2+1)	3691 (0.05)	160701 (2.28)	3756559 (53.32)
Exponential (2^t)	2875 (0.05)	107515 (1.53)	4131611 (58.64)

Table 7.1: Clique-tree costs for sample networks

for $i = 1$ to $n-1$, and the size of the corresponding cross product space is the product of $|\Omega_{(S_R,i)}|$, $|\Omega_{(S_R,i+1)}|$, $|\Omega_{(S_T,i)}|$, and $|\Omega_{(S_T,i+1)}|$. For fixed state spaces, this product is just $|\Omega_{S_R}|^2|\Omega_{S_T}|^2$. However, if we restrict the state spaces for the spatial variables on the basis of some initial location estimate and some bounds on how quickly the robot and the target can move about, we can do considerably better.

Table 7.1 shows the clique-tree costs for three MTL decision model networks of size $n = 3, 5$, and 8 time points. For each size of model, we consider cases in which $\Omega_{(S_R,i)}$ and $\Omega_{(S_T,i)}$ are constant for all $1 \geq i \geq n$, and cases in which $|\Omega_{(S_R,1)}| = |\Omega_{(S_T,1)}| = 1$ and the sizes of the state spaces for subsequent spatial variables, $\Omega_{(S_R,i)}$ and $\Omega_{(S_T,i)}$, for $1 > i \geq n$ grow by linear, quadratic, and exponential factors bounded by $|\Omega_{S_T}| = |\Omega S_R| = 30$. For these evaluations, $|\Omega_{A_R}| = 6$, $|\Omega_{O_T}| = 32$, and $|\Omega_{O_R}| = 19$ in keeping with the sensory and movement routines of our current robot. The number in brackets underneath the clique tree cost is the time in cpu seconds required for evaluation.

Our current idea for restricting the present location of the robot and the target involves using a fixed threshold and the most up-to-date estimates for these locations to eliminate unlikely possibilities. Occasionally, the actual locations will be mistaken'y eliminated, and the robot will fail to track the target. There will have tu be a recovery strategy and a criterion for invoking it to deal with such failures.

There are certain costs involved with evaluating Bayesian networks that we have ignored so far. These costs involve triangulating the graph, constructing the clique tree, and performing the storage allocation for building the necessary data structures. For our approach of dynamically restricting the range of spatial variables, the state spaces for the random variables change, but the sizes of these state spaces and the topology of the Bayesian network remain constant. As a consequence, these ignored costs are incurred once, and the associated computational tasks can be carried out at design time. Dynamically adjusting the state spaces for the spatial variables is straightforward and computationally inexpensive.

The third method for reducing the cost of decision making involves reducing the size of $\mathcal{S}$, the set of sequences of actions corresponding to tracking and localization strategies. For an n step lookahead, the set of useful strategies of length n or less is a very small subset of $\Omega_{A_R}{}^n$. Still, given that we have to evaluate the network $|\mathcal{S}|$ times, even a relatively small $\mathcal{S}$ can cause problems. To reduce $\mathcal{S}$ to an acceptable size, we only evaluate the network for strategies that are possible given the current restrictions on the spatial variables. For instance, if the robot knows that it is moving down a corridor toward a left-pointing L junction, it can eliminate from consideration any strategy that involves it moving to the end of the corridor and turning right. With appropriate preprocessing, it is computationally simple to dynamically reduce $\mathcal{S}$ to just a few possible strategies in most cases.

7.6 Further Reading

Bayesian decision theory [5, 8, 33]. Value of information [19]. It should be noted that Howard's is not the only theory proposed for assessing the value of information sources. In particular, information value theory is closely related to the theory of experimental design [16, 30]. Experimental design is concerned with the problem of maximizing the information gained from performing experiments under cost constraints. Information value theory represents one approach to experimental design based on Bayesian decision theory.

Influence diagrams [20]. Dynamic programming [7]. Conditioning [18]. Keiji's join-tree cost [22]. Jensen's [21] variation on Lauritzen and Spiegelhalter's clustering algorithm [25]. Causal poly trees [32]. Evaluating influence diagrams [34]. Influence diagrams for control applications [1].

The notion of locally distinctive place as it is used in Section 7.3 is due to Kuipers [23]. The design of the geographer module was based on the work of Kuipers [24] and Levitt [26] on learning maps of large-scale space. and the extensions of Basye *et al* [6] to handle uncertainty.

See Dean and Kanazawa [13] and Hanks [17] for competing approaches. See Cooper *et al* [10] for a discussion of a related approach to probabilistic reasoning about change using a discrete model of time.

References to work on active perception [2, 3, 4].

Bibliography

[1] Agogino, A. M. and Ramamurthi, K., *Real-Time Influence Diagrams for Monitoring and Controlling Mechanical Systems*. Technical report, Department of Mechanical Engineering, University of California, Berkeley, 1988.

[2] Aloimonos, J., Bandyopadhyay, A., and Weiss, I., Active Vision, *Proceedings of the First International Conference on Computer Vision*, 1987, 35–55.

[3] Bajcsy, R., Active Perception, *Proceedings of the IEEE*, **76**(8) (1988) 996–1005.

[4] Ballard, Dana H., Reference Frames for Animate Vision, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 1635–1641.

[5] Barnett, V., *Comparative Statistical Inference*, (John Wiley and Sons, New York, 1982).

[6] Basye, Kenneth, Dean, Thomas, and Vitter, Jeffrey Scott, Coping With Uncertainty in Map Learning, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 663–668.

[7] Bellman, Richard, *Dynamic Programming*, (Princeton University Press, 1957).

[8] Chernoff, Herman and Moses, Lincoln E., *Elementary Decision Theory*, (John Wiley and Sons, New York, 1959).

[9] Cooper, Gregory F., *Probabilistic Inference Using Belief Networks is NP-Hard*, Technical Report KSL-87-27, Stanford Knowledge Systems Laboratory, 1987.

[10] Cooper. Gregory F., Horvitz. Eric J., and Heckerman. David E., *A Method for Temporal Probabilistic Reasoning.* Technical Report KSL-88-30. Stanford Knowledge Systems Laboratory, 1988.

[11] Dean, Thomas, Basye, Kenneth, Chekaluk, Robert, Hyun, Seungseok, Lejter. Moises, and Randazza, Margaret. Coping with Uncertainty in a Control System for Navigation and Exploration, *Proceedings AAAI-90, Boston. Massachusetts*, AAAI. 1990. 1010–1015.

[12] Dean. Thomas, Basye, Kenneth, and Lejter. Moises, Planning and Active Perception, *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, DARPA. 1990.

[13] Dean. Thomas and Kanazawa. Keiji, Probabilistic Temporal Reasoning, *Proceedings AAAI-88. St. Paul. Minnesota.* AAAI. 1988. 524–528.

[14] Dean, Thomas and Kanazawa. Keiji, A Model for Reasoning About Persistence and Causation, *Computational Intelligence*, 5(3) (1989) 142–150.

[15] Dean, Thomas and Kanazawa, Keiji, Persistence and Probabilistic Inference, *IEEE Transactions on Systems, Man, and Cybernetics*, 19(3) (1989) 574–585.

[16] Fedorov, V., *Theory of Optimal Experimental Design*, (Academic Press, New York, 1972).

[17] Hanks, Steve, Representing and Computing Temporally Scoped Beliefs, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 501–505.

[18] Horvitz, Eric J.; Suermondt, H. Jacques, and Cooper, Gregory F., *Bounded Conditioning: Flexible Inference for Decisions Under Scarce Resources*, Technical Report KSL-89-42, Stanford Knowledge Systems Laboratory, 1989.

[19] Howard, Ronald A., Information Value Theory. *IEEE Transactions on Systems Science and Cybernetics.* 2(1) (1966) 22–26.

[20] Howard, Ronald A. and Matheson, James E., Influence Diagrams, Howard, Ronald A. and Matheson, James E., (Eds.). *The Principles and Applications of Decision Analysis*, (Strategic Decisions Group, Menlo Park, CA 94025, 1984).

[21] Jensen, Finn V., Lauritzen, Steffen L., and Olesen, Kristian G., *Bayesian Updating in Recursive Graphical Models by Local Computations*. Technical Report R 89-15, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg, 1989.

[22] Kanazawa, Keiji, *Probability, Time, and Action*, PhD thesis, Brown University, Providence, RI, Forthcoming.

[23] Kuipers, Benjamin, Modeling Spatial Knowledge, *Cognitive Science*, **2** (1978) 129–153.

[24] Kuipers, Benjamin J. and Byun, Yung-Tai, A Robust, Qualitative Method for Robot Spatial Reasoning, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 774–779.

[25] Lauritzen, Stephen L. and Spiegelhalter, David J., Local computations with probabilities on graphical structures and their application to expert systems, *Journal of the Royal Statistical Society*, **50**(2) (1988) 157–194.

[26] Levitt, Tod S., Lawton, Daryl T., Chelberg, David M., and Nelson, Philip C., Qualitative Landmark-based Path Planning and Following, *Proceedings AAAI-87, Seattle, Washington*, AAAI, 1987, 689–694.

[27] McCarthy, John, Applications of Circumscription to Formalizing Commonsense Knowledge, *Artificial Intelligence*, **28** (1986) 89–116.

[28] McCarthy, John and Hayes, Patrick J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, *Machine Intelligence*, **4** (1969) 463–502.

[29] McDermott, Drew V., A temporal logic for reasoning about processes and plans, *Cognitive Science*, **6** (1982) 101–155.

[30] Mendenhall, W., *Introduction to Linear Models and the Design and Analysis of Experiments*, (Wadsworth, Belmont, California, 1968).

[31] Neapolitan, Richard E., *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*, (John Wiley and Sons, New York, 1990).

[32] Pearl, Judea, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, (Morgan-Kaufmann, Los Altos, California, 1988).

[33] **Raiffa.** Howard and Schlaifer. R.. *Applied Statistical Decision Theory.* (Harvard University Press. 1961).

[34] Shachter. Ross D.. Evaluating Influence Diagrams. *Operations Research.* **34**(6) (1986) 871–882.

[35] Syski. Ryszard. *Random Processes.* (Marcel Dekker. New York. 1979).

Chapter 8

Controlling Inference

This chapter describes approaches for designing systems that are capable of taking their own computational resources into consideration during planning and problem solving. In particular, we are interested in systems that manage their computational resources by using expectations about the performance of decision making procedures and preferences over outcomes resulting from applying such procedures. Careful management of computational resources is important for complex problem solving tasks in which the time spent in decision making affects the quality of the responses generated by a system.

Much of the work described in this chapter can be seen as a response to a movement, started in the early 1980's, away from systems that make use of complex representations and engage in lengthy deliberations, and towards systems capable of making many very simple decisions quickly. This movement brought about the advent of the so-called "reactive systems" described in Chapter 4. Most reactive systems are essentially programming languages for building systems that must be responsive to their environment. Such languages generally allow for multiple asynchronous decision processes, facilitate communication among processes, and provide support for interrupts and process arbitration.

Many of the researchers building reactive systems were interested in robotics and decision-support applications requiring real-time response. The responsiveness of reactive systems was in stark contrast with the performance of most planning and problem solving systems in use at that time. Most existing planning systems were essentially off-line data processing procedures that accepted as input some initial (and generally complete) descrip-

[0] ©1990 Thomas Dean. All rights reserved.

tion of the current state of the environment, and, after some indeterminate (and generally lengthy) delay, returned a rigid sequence of actions which, if the environment was particularly cooperative, might result in the successful achievement of some goal.

Reactive systems might be seen as an extreme response to the shortcomings of the existing planning systems. Reactive systems provided responsiveness at the cost of shallow and often short-sighted decision making. Since there were no proposals for how to control decision making in time-critical situations, researchers turned away from the traditional approaches to planning and attempted to incorporate more sophisticated decision making into reactive systems. Unwilling to sacrifice response time, the researchers that were trying to improve the decision-making capabilities of reactive systems were forced to trade space for time, often without a great deal of attention to the consequences.

Some of the dissatisfaction with complex representations and complicated deliberation strategies was due to misinterpreting asymptotic complexity results as evidence of the existence of impassable computational barriers. Proofs of NP-hardness certainly indicate that we must be prepared to make concessions to complexity in the form of tradeoffs. The lesson to be learned, however, is that we have to control inference, and not that we have to abandon it altogether.

In the 1970's, a great deal of effort was spent studying systems capable of explicitly reasoning about their own decision-making capabilities. This sort of reasoning about reasoning is generally referred to as *meta-reasoning*. As the research in this area matured, some researchers were concerned with how to learn to control decision making, while others were interested in the basic mechanisms required to guide decision making under time pressure. Many of the mechanisms studied had in common the use of expectations regarding the performance of decision procedures to help in selecting from among a set of such decision procedures.

As researchers began looking in the literature, it became clear that many of the tools required for reasoning about the costs and benefits of applying decision-making routines were already available. Indeed, researchers in the decision sciences had already considered some of the problems involved in reasoning about the costs and benefits of inference. However, with rare exception,[1] the decision analysts assumed that the agent was possessed of

[1] I. J. Good was one of those exceptions, and, in an amazingly forward looking paper [22], Good talked about what he called *type II rationality* which involves an agent reasoning

unlimited computational capabilities for reasoning about its current knowledge; the issue most often addressed concerned whether or not an agent should consider adding to its current knowledge. We are interested in the case of an agent currently biased to act in a certain way and considering if it should expend further computational resources and risk the consequences of delay in order to deliberate further about its options. It is this basic idea of an agent with limited computational capabilities, embedded in a complex environment with other agents and processes not under its control, and reasoning about the costs and benefits of continued deliberation that is the subject of this chapter.

8.1 Decision Theory and the Control of Inference

We begin with the idea of a *decision procedure*: a procedure used by an agent to select an action which, if executed, changes the world. Some actions are purely computational. For our purposes, such computational actions correspond to an agent running a decision procedure, and we refer to such actions as *inferential*. The results of inferential actions have no immediate effect on the world external to the agent, but they do have an effect on the internal state of the agent. In particular, inferential actions consume computational resources that might be spent otherwise, and generally result in the agent revising its estimations regarding how to act in various circumstances. In addition to the purely computational actions, there are *physical actions* that change the state of the world external to the agent, but that may also require some allocation of computational resources.

Real agents have severely limited computational capabilities, and, since inferential actions take time, an inferential action may end up selecting a physical action that might have been appropriate at some earlier time, but that is no longer so. Inferential actions are useful only insofar as they enable an agent to select physical actions that lead to desirable outcomes.[2] Decision theory provides us with the language required to talk precisely about what it would mean for an action to lead to a desirable outcome. Before we can proceed further, we will need some precise language for talking about possible outcomes and stating preferences among them. The language that

about its own abilities to reason.

[2] Inferential actions are also useful for learning purposes, as in learning search strategies, but even here the actions are ultimately in service to selecting physical actions that lead to desirable outcomes.

we adopt is borrowed directly from statistical decision theory.

~~*The following paragraphs should be subsumed by the discussion of decision theory in Chapter 7, but they remain until that chapter is further along.*~~

~~In the simplest case, we might consider an agent faced with choosing from among a set of completely defined and immediately attainable alternatives (*e.g.*, a student might be faced with choosing between seeing a movie or studying for an exam). The agent might ignore some of the implications of its actions and focus on immediate rewards (*e.g.*, a relaxing respite from work or an increase in knowledge about a given subject), but, more often than not, the agent will be concerned with the long-term implications or *consequences* of its actions (*e.g.*, the possibility of achieving a high score on an exam which in turn might raise the chances of getting into graduate school). In general, we cannot guarantee these consequences; they are seldom immediately attainable and they are usually only partially defined. If we ignore the long-term implications of our actions, the alternatives can be viewed as rewards, and a rational agent would simply choose the reward that it considers best. In the case in which the agent is concerned about the consequences of its actions and those consequences are not entirely under its control, the picture is more complicated. In this case, the agent might have a probability distribution over the set of possible consequences and some way of assigning values to the individual consequences so as to form expectations regarding the value of the possible outcomes or *prospects* resulting from performing alternative actions.~~

Talk about the value of computation and contrast it with the utility function

Let Ω correspond to a set of possible states of the world. We assume that the agent has a function,

$$U : \Omega \rightarrow \mathbf{R};$$

that assigns a real number to each state of the world. This is referred to as the agent's *utility function*.[3] These numbers enable the agent to compare various states of the world that might result as a consequence of its actions. It is assumed that a rational agent will act so as to maximize its utility. The quantity, $U(\omega)$ where $\omega \in \Omega$, is generally meant to account for both the immediate costs and benefits of being in the state ω and the delayed costs and benefits derived from future possible states. We assume that there is some process deterministic or stochastic governing the transition between

[3] See Chernoff and Moses [9], Barnett [2], or Pearl [37] for discussions regarding the axioms of utility theory.

states, and that this process is partially determined or biased by the agent's choice of action. In the case of a stochastic process, the agent cannot know what state will result from a given action and hence the agent must make use of expectations regarding the consequences of its actions. In order to account for these longer-term consequences, it is often useful to think of the agent as having a particular long-term plan or *policy*. In such cases, the agent will generally assign an expected utility to a given state based upon the immediate rewards available in that state and expectations about the subsequent states, given that the agent continues to select actions based upon its current policy.

In addition to expectations about the possible future consequences of its physical actions, an agent capable of reasoning about its computational capabilities must also have expectations regarding the potential value of its computations, and estimates of how long those computations are likely to take. In most of the work discussed in this paper, an agent is assumed to engage in some sort of meta-reasoning. For our purposes, meta-reasoning consists of running a decision procedure whose purpose it is to determine what other decision procedures should run and when. We prefer the term *deliberation scheduling* [13] to the more general meta-reasoning and will use the two interchangeably in this paper. If the meta-level decision procedure takes a significant amount of time to run, it must be argued that this time is well spent. In some cases, the time spent in meta-reasoning is small enough that it can be safely ignored; in other cases, it may be useful to invoke a meta-meta-level decision procedure to reason about the costs and benefits of meta-reasoning.

Refer back to the material in Chapter 7 on the value of information.

Note that so far we have not accounted for the computational cost of *deliberating* about the value of a particular information source. In information processing systems, information costs in terms of the time and resources expended in computing an answer to a query. Neither have we closely considered how an agent might compute an expectation such as $E[r|\mathcal{E}]$. We may know how to compute such an expectation, but it may be that an agent can not afford to compute it. In the following sections, we build on the basic idea behind information value theory to account for systems that have limited computational capabilities.

The rest of this chapter is organized as follows. In Section 8.2, we consider a general approach to studying the control of reasoning that casts the general problem in terms of search. In this same section, we also investigate some of the practical issues that constrain how an agent might reason about its computational capabilities: these constraints and the measures taken to deal with them apply to all of the work discussed in this chapter. Section 8.3 considers an approach to reasoning about computational capabilities that relies on a particular class of algorithms for implementing decision procedures. Section 8.5 briefly considers some related issues in design-time meta-reasoning for compiling run-time systems for time-critical applications.

8.2 Control of Problem Solving

In this section. we consider a general approach to reasoning about decision-theoretic control of inference due to Russell and Wefald [40. 41]. As in most decision problems. the basic goal is for the agent to maximize its utility function $U(\omega)$ on states of the world $\omega \in \Omega$. We assume that the agent has some set of *base-level* actions $\mathcal{A}$ that it can execute to affect its environment. Borrowing Russell and Wefald's notation. we denote the outcome of an action A performed in state ω as $[A.\omega]$ or just $[A]$ if the action is performed in the current state.

At any given time the agent has a default action $\alpha \in \mathcal{A}$ which is the action that currently appears to be best. In addition. the agent has a set of computational actions $\{S_i\}$ which might cause the agent to revise its default action. The agent is faced with the decision to choose from among the available options: $\alpha, S_1, S_2, \ldots, S_k$. Computational actions only affect the agent's internal state. However. time passes while the agent is deliberating and opportunities are lost, so the net value of computation is defined to be the difference between the utility of the state resulting from the computation minus the utility of the state resulting from executing α.

$$V(S_j) = U([S_j]) - U([\alpha]).$$

If the computation S_j results in a revised assessment of the best action, α_{S_j}, and a commitment to perform this action, then

$$U([S_j]) = U([\alpha_{S_j}, [S_j]]).$$

where $[\alpha_{S_j}. [S_j]]$ indicates the outcome of the action α_{S_j} in the state following the computation S_j. Alternatively. if S_j is a partial computation (*i.e.*. a computation that doesn't immediately result in a revised assessment of a best action. but that provides intermediate results leading to a revised assessment), then

$$U([S_j]) = \sum_{T} \Pr(T)U([\alpha_T.[S_j.T]]),$$

where T ranges over all possible complete computations following S_j, $S_j.T$ denotes the computation corresponding to S_j immediately followed by T, and $\Pr(T)$ is the probability that the agent will perform the computation T.

Generally, the agent doesn't know the exact utilities or probabilities, and so it must compute an estimate using some amount of its computational resources. Let $\hat{Q}^S$ denote the estimate of the quantity Q following a computation S. In this case, we have

$$\hat{U}^{S.S_j}([S_j]) = \sum_{T} \hat{\Pr}^{S.S_j}(T)\hat{U}^{S.S_j}([\alpha_T.[S_j.T]]),$$

where S is the total computation prior to considering S_j, and

$$\hat{U}^{S.S_j}([\alpha_T.[S_j.T]]) = \max_i \hat{U}^{S.S_j}([A_i.[S_j.T]]),$$

where the A_i range over all possible base-level actions in $\mathcal{A}$. By superscripting quantities to indicate the computations required to generate the corresponding estimates, Russell and Wefald are attempting to capture the behavior of real agents with realistically limited computational capabilities. At each point in time, the agent decides how to act based upon whatever estimates it currently has, using a meta-reasoning decision procedure whose time cost is assumed to be negligible. The meta-reasoning decision procedure is responsible for deciding whether further deliberation is warranted, and it does so on the basis of the estimated net value of computation,

$$\hat{V}^{S.S_j}(S_j) = \hat{U}^{S.S_j}([S_j]) - \hat{U}^{S.S_j}([\alpha]).$$

As Russell and Wefald point out, before the computation S_j is performed $\hat{V}(S_j)$ is just a random variable, and so the agent, not knowing the exact value, computes an expectation,

$$E[\hat{V}^{S.S_j}(S_j)] = E[\hat{U}^{S.S_j}([S_j]) - \hat{U}^{S.S_j}([\alpha])]. \tag{8.1}$$

It is worth noting the difference between Equation 8.1 and the following equation introduced in Chapter 7 in presenting Howard's value of information theory,

$$E(V(I_X)|\mathcal{E}) = E(V([])|I_X, \mathcal{E}) - E(V([])|\mathcal{E}).$$

The important difference is that both terms in the right-hand-side expectation in Equation 8.1 change as a consequence of further inference. If an agent had unlimited computational capabilities, it would not be computing estimates, and only the first of the two terms would require expending computational resources since it would be the case that

$$\hat{U}^{S.S_j}([\alpha])] = \hat{U}^{S}([\alpha])].$$

However, we are concerned with agents with limited computational capabilities, and further computation will likely result in a better estimate of the utilities for $[\alpha]$ as well as for $[\alpha', [S_j]]$ for any action $\alpha' \in \mathcal{A}$.

In order to simplify reasoning about the utility of combined computational and base-level actions, Russell and Wefald separate the *intrinsic utility*, that is the utility of an action independent of time, from the *time cost* of computational actions, defining the utility of a state as the difference between these two:

$$\hat{U}([A_i, [S_j]]) = \hat{U}_I([A_i]) - TC(S_j).$$

It should be noted, however, that determining an appropriate time cost function can become quite complicated in applying Russell and Wefald's approach. In particular, costs concerning hard and soft deadlines will have to be accounted for by this function. In the game-playing application explored in [40], there are no hard deadlines on a per-move basis, instead there is a per-game time limit that is factored into the time cost. In many time-critical problem solving applications, calculating the time cost can be quite complicated (*e.g.*, consider the sort of medical care applications investigated in [28] and [25]). Russell and Wefald assume that the time cost is independent of both the computation itself and its recommendations. The former is certainly reasonable, and, since the recommendation is not known at meta-reasoning time, the latter is also reasonable. However, one could easily imagine employing an expected time cost based on some *a priori* knowledge concerning possible recommendations. It should be noted that all of the approaches described in this chapter make assumptions about time cost similar to those of Russell and Wefald.

Perhaps the nicest part of the Russell and Wefald work is their careful treatment of the criterion for deciding whether or not to expend further resources on deliberation. If the agent is considering at most one additional computational step, then it is only interested in computations that serve to update the expected value of a given base-level action so as to supplant the current default action. The expected gain from a given computation is measured in terms of the difference between the current expectation regarding one action and the anticipated revised expectations regarding a second action where one of the two actions is the default action. Intuitively, further deliberation is called for whenever the difference between the expected gain in utility from a computation and the associated cost of delay is greater than zero. Russell and Wefald identify two cases to consider in deciding to perform a computation aimed at providing a revised assessment of the best action to perform. In the first case, we suppose that there exists a computation S_j which affects the agent's estimation of the utility of the alternative action β so that

$$E[V(S_j)] = \int_{\hat{U}_I([\alpha])}^{\infty} p_{\beta,j}(x)(x - \hat{U}_I^{S}([\alpha]))dx - TC(S_j), \tag{8.2}$$

where $p_{\beta,j}$ is the probability density function for $\hat{U}_I^{S.S_j}([\beta])$. In the second case, there exists a computation S_k which affects the agent's estimation of the utility of the default action α so that

$$E[V(S_k)] = \int_{-\infty}^{\hat{U}_I([\beta])} p_{\alpha,k}(x)(\hat{U}_I^{S}([\beta]) - x)dx - TC(S_k), \tag{8.3}$$

where $p_{\alpha,k}$ is the probability density function for $\hat{U}_I^{S.S_k}([\alpha])$. If there are n computational actions and m base-level actions, then each meta-level reasoning step will require computing each of Equations 8.2 and 8.3 nm times. If the distributions governing utility estimates are simple in form (*e.g.*, normal distributions), computing the integrals in Equations 8.2 and 8.3 can be done quite efficiently.

There are a number of assumptions that Russell and Wefald make in their analysis. First, it is assumed that the agent considers only single computation steps, estimates their ultimate effect, and then chooses the step appearing to have highest benefit. This is referred to as the *meta-greedy assumption*. Second, it is assumed that the agent will act as though it will take at most one more search step. This is referred to as the *single-step assumption*. Finally, it is assumed that a computational action will change the expected utility estimate for exactly one base-level action. In Russell and Wefald's state-space search paradigm, this is referred to as the *subtree-independence* assumption.

The assumptions stated in the previous paragraph may seem overly restrictive, but it is quite difficult to avoid these or similar assumptions in general. Pearl [37] identifies two assumptions that most practical meta-reasoning systems ascribe to: *no competition*, each information source is evaluated in isolation from all the others, and *one-step horizon*, we consult at most one additional information source before committing to a base-level action. Assessments of information sources based on the no-competition and one-step-horizon assumptions are referred to as *myopic*, and most practical systems employ myopic decision policies.

The next piece of research that we consider in this section is due to Etzioni [18], and it borrows from the Russell and Wefald work, and builds on the early work of Simon and Kadane [43] on satisficing search. It is particularly interesting for the fact that it attempts to combine the sort of goal-driven behavior prevalent in artificial intelligence with the decision theoretic view of maximizing expected ~~utility~~. In Etzioni's model, the agent is given a set of goals $\{G_1, G_2, \ldots, G_n\}$, a set of methods M, and a deadline B. The agent attempts to determine a sequence of methods

globally replace util by value

$$\sigma = m_{1,1}, m_{2,1}, \ldots, m_{k_1,1}, m_{1,2}, m_{2,2}, \ldots, m_{k_2,2}, \ldots, m_{1,n}, m_{2,n}, \ldots, m_{k_n,n}$$

where $m_{i,j}$ is the ith method to be applied to solving the jth goal. The idea is that the agent will apply each method in turn until it either runs out of methods or achieves the goal, at which point it will turn its attention to the next goal. The expected ~~utility~~ of σ is

$$\begin{aligned}
\mathrm{E}[\hat{\mathrm{U}}(\sigma)] \;=\;& \mathrm{E}[\hat{\mathrm{U}}(m_{1,1})] + \cdots + \mathrm{E}[\hat{\mathrm{U}}(m_{k_1,1})] \prod_{i=1}^{k_1-1} (1 - \Pr(m_{i,1})) + \\
& \mathrm{E}[\hat{\mathrm{U}}(m_{1,2})] + \cdots + \mathrm{E}[\hat{\mathrm{U}}(m_{k_2,2})] \prod_{i=1}^{k_2-1} (1 - \Pr(m_{i,2})) + \\
& \vdots \\
& \mathrm{E}[\hat{\mathrm{U}}(m_{1,n})] + \cdots + \mathrm{E}[\hat{\mathrm{U}}(m_{k_n,n})] \prod_{i=1}^{k_n-1} (1 - \Pr(m_{i,n})).
\end{aligned}$$

In this simple model, no provision is made for switching back and forth between goals, and, except for ignoring the remaining methods for a given goal once that goal has been achieved, no provision is made for modifying the search as new information becomes available. Etzioni defines the expected *opportunity cost* (γ_B) of a method m for a deadline B as

$$\mathrm{E}[\gamma_B(m)] = \mathrm{E}[\hat{\mathrm{U}}(\sigma^*_B)] - \mathrm{E}[\hat{\mathrm{U}}(\sigma^*_{B-TC(m)})].$$

where σ_B^* is the optimal method sequence for a deadline B. and $TC(m)$ is the expected time to carry out method m. In addition. the expected *gain* (G_B) of a method m for a deadline B is defined to be

$$\mathrm{E}[G_B(m)] = \mathrm{E}[\dot{\mathrm{V}}(m)] + \mathrm{E}[\gamma_B(m)].$$

He then shows that by repeatedly choosing the method whose expected gain is maximal an agent will construct an optimal method sequence.

From one point of view, Etzioni's work is not about meta-reasoning at all; his work is concerned with ordinary sequential decision problems. For these problems. Etzioni points out that. in certain cases, the cost of determining an optimal method sequence can be quite high. In other words. we can't ignore the cost of meta-level reasoning in the decision-making model. His analysis showing that sorting methods on their marginal utility can often result in optimal or near-optimal method sequences is exactly the sort of analysis required to justify a particular meta-level reasoning.

In the case in which the agent has a single goal and multiple methods for achieving it. the requisite meta-reasoning is easy. In particular. suppose that there is a constant opportunity cost γ per unit of time spent on the goal. and for each method $m \in M$ the agent has an expected time cost $\mathrm{E}[TC(m)]$. an expected utility estimate $\mathrm{E}[\hat{\mathrm{U}}(m)]$, and a probability $\Pr(m)$ of achieving the goal using that method. The expected gain of a method m is just

$$\mathrm{E}[G(m)] = \mathrm{E}[\hat{\mathrm{U}}(m)] - \gamma \mathrm{E}[TC(m)],$$

and the task is to find σ so as to maximize

$$\mathrm{E}[\hat{\mathrm{U}}(\sigma)] = \sum_{i \text{ s.t. } \mathrm{E}[G(m_i)]>0} \mathrm{E}[G(m_i)] \prod_{k=0}^{i-1} (1 - \Pr(m_k)).$$

Etzioni claims, and it is easy to verify, that. by sorting the methods in increasing order using $\frac{\mathrm{E}[G(m)]}{\Pr(m)}$ as a key, an agent can construct an optimal method ordering.

In the above case, it is plausible to assume that the cost of meta-reasoning (*i.e.*. the time spent calculating $\frac{\mathrm{E}[G(m)]}{\Pr(m)}$ for each method m and sorting using the results) is negligible in comparison with the cost of applying a method. In the case of an agent faced with multiple goals even where there is only one method for each goal. it is more difficult to make such an assumption. By reducing the *knapsack problem* [20] to the problem of computing the expected opportunity cost. Etzioni shows that computing the expected opportunity cost of a method is NP-complete.

It is not too surprising that there are some hard problems lurking among the deliberation scheduling problems that underlie decision-theoretic control of inference. It should be pointed out, however, that all we should really be concerned with is the expected cost of meta-reasoning, and that, in many practical applications, approximations are much preferred to even polynomial-time methods for computing exact solutions.

Etzioni suggests using Garey and Johnson's factor-of-two approximation [20], but it should be noted that the knapsack problem is a number problem for which there exist pseudo-polynomial time algorithms and good branch-and-bound approximations. These branch-and-bound algorithms have exponential-time worst-case behavior, but their expected performance is such that many practitioners consider knapsack tractable. In the next section, we see how approximation algorithms for computationally expensive problems can provide us with even greater flexibility in allocating processor time to decision procedures.

While Etzioni's invocation of asymptotic complexity as a measure of difficulty may not be particularly appropriate in this case, it does force the reader to reconsider the assumptions regarding the time cost of meta-reasoning. For instance, if n is small and the average time cost of the methods in M is high, then it may even be reasonable to perform a meta-computation whose worst-case behavior is exponential in n. It may even be useful to add another level of meta-reasoning to reason about various alternative scheduling algorithms.

Just what is the structure of the decision making process that we are seeking to control? In the Russell and Wefald model, object-level decision making involves fixed-duration computations that attempt to provide a better assessment of a single base-level action. In the next section, we consider problems in which the meta-level reasoner sacrifices some of its control over object-level decision making in order to simplify meta-level decision making. In particular, we consider decision procedures that return estimates that improve with additional allocations of processor time. The ability to preempt decision procedures at any time during their computation simplifies deliberation scheduling in many cases.

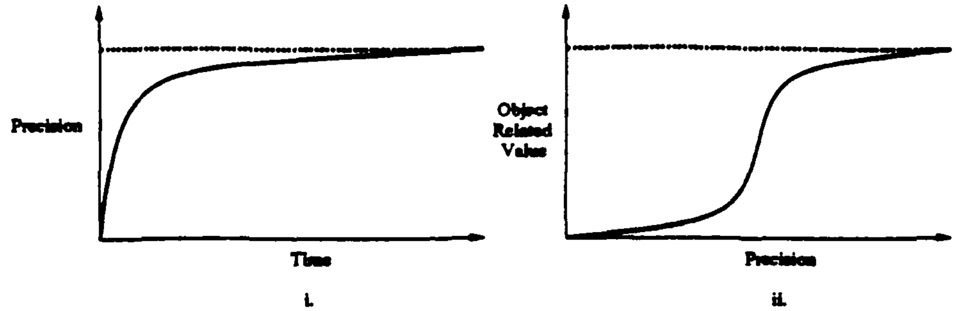


Figure 8.1: Plots relating (i) time spent in computation to the precision of a probabilistic calculation, and (ii) precision to the value associated with getting the diagnosis correct and treating the patient accordingly (after [27]).

(Pliant, Tractable)
(Reasoning)
Flexible Inference Strategies

8.3 Flexible Computations and Anytime Algorithms

In this section, we consider two independently developed but closely related approaches to decision-theoretic control of problem solving. The two approaches are due to Dean and Boddy [13] and Horvitz [27]. Horvitz refers to his decision procedures as *flexible computations* and Dean and Boddy refer to theirs as *anytime algorithms*, but the basic idea behind the two proposals is the same, and we will use the two terms interchangeably.

In the ideal flexible computation, the object-related value of the result returned by a decision procedure is a continuous function of the time spent in computation. The notion of "object-related value" of a result is to be contrasted with the "comprehensive value" of a system's response to a given state; the latter refers to the overall utility of the response and the former is some measure of the value of the result apart from its use in a particular set of circumstances. Object-related value is exactly Russell and Wefald's intrinsic utility. In some cases, the task of relating a result to the comprehensive value of the overall response can be quite complex. This is especially so in cases in which there are several results from several different decision procedures. In these cases, it is often convenient to make the assumption that the value function is *separable* so that the comprehensive value of the system's response can be computed as the sum of the value of a sequence of outcomes.

We assume that a flexible computation can be interrupted at any point

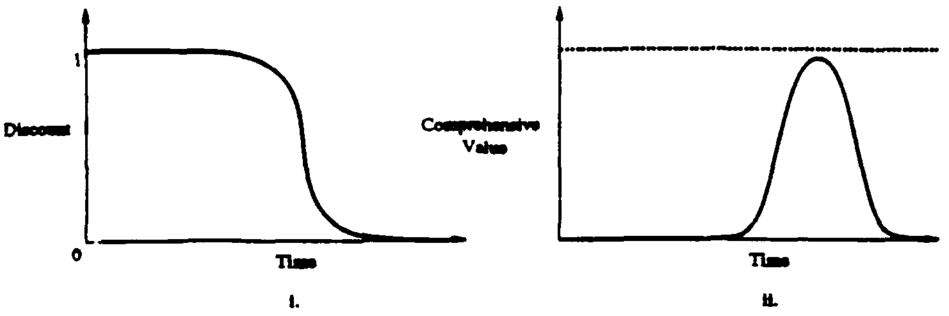


Figure 8.2: Plots indicating (i) a discounting factor for delayed treatment, and (ii) the comprehensive value of computation as a function of time (after [27]).

during computation—hence the name "anytime"—to return an answer whose object-related value increases as it is allocated additional time. Horvitz provides a good example of a flexible computation and an analysis of its object-related value drawn from the health care domain. Suppose that we have an anytime algorithm that computes a posterior distribution for a set of possible diagnoses given certain information regarding a particular patient. Figure 8.1.i ~~(from [27])~~ shows a graph that relates the precision of the result returned by this algorithm to the time spent in computation. The object-related value can be determined as a function of precision by considering the expected utility of administering a treatment based on a diagnosis of a given precision ignoring when the treatment is administered (see Figure 8.1.ii).

more on contributions

The comprehensive value of computation is meant to account for the costs and benefits related to the time at which the results of decision procedures are made use of to initiate physical actions. Figure 8.2.i (from [27]) indicates how a physician might discount the object-related value of a computation as a function of delay in administering treatment. The comprehensive value of computation is shown in Figure 8.2.ii and is obtained by combining the information in Figures 8.1.ii and 8.2.i. This method of combining information assumes both time-cost separability and one-step horizon.

Both Horvitz and Dean and Boddy note that the most useful sort of flexible computations are those whose object-related value increases monotonically over some range of computation times. Dean and Boddy [13] employ decision procedures that are monotonic throughout the range of computation times, and exploit this fact to expedite deliberation scheduling for a special class of planning problems referred to as *time-dependent planning problems*. A planning problem is said to be time-dependent if the time available for responding to a given event varies from situation to situation. In their model, a predictive component, whose time cost is not considered, predicts events and their time of occurrence on the basis of observations, and the planning system is given the task of formulating a response to each event and executing it in the time available before the event occurs.

TDP defined

The model of Dean and Boddy generalizes on the multiple-goals/single-method-for-each model of Etzioni described in the previous section, by allowing each goal to have a separate deadline. If the responses to the different events are independent, the task of deliberation scheduling can be stated in terms of maximizing the sum

$$\sum_{e \in E} V(\mathit{Response}(e)),$$

expected value

where E is the set of all events predicted thus far. It is assumed that there is exactly one decision procedure for each type of event likely to be encountered, and that there are statistics on the performance of these decision procedures. The statistics are summarized in what are called *performance profiles* which are essentially the same as the graphs used by Horvitz in his analysis (*e.g.*, see Figure 8.1.ii).

In Section 8.4, we define the class of time-dependent planning problems precisely, and provide polynomial-time algorithms for deliberation scheduling for particular subclasses. These algorithms use a simple greedy strategy working backward from the last predicted event, choosing the decision procedure whose expected gain computed from the performance profiles is greatest.

meta-meta-level
meta-level
base-level
↑

It is worth considering why the NP-completeness result reported by Etzioni does not apply in this more general case. In job-shop scheduling, if it is possible to suspend, and later resume, a job, then many otherwise difficult problems become trivial [23, 5]. Such (preemptive) scheduling problems are somewhat rare in real job shops given that there is often significant overhead involved is suspending and resuming jobs (*e.g.*, traveling between workstations or changing tools), but they are considerably more common with regard to purely computational tasks (*e.g.*, suspending and resuming Unix processes). In many scheduling problems, each job has a fixed cost and requires a fixed amount of time to perform: spending any less than the full amount yields no advantage. This is the case in the decision procedures considered by Etzioni. If, however, the decision procedures for computing appropriate actions are preemptible and provide better answers depending upon the time available for deliberation, then the task of deliberation scheduling is considerably simplified. Anytime decision procedures thus provide more flexibility in responding to time-critical situations, and simplify the task of allocating processor time in cases where there is contention among several decision tasks.

For the multiple-goals/single-method-for-each problem described in the previous section, Etzioni suggests using a factor-of-two approximation to avoid potential combinatorics in deliberation scheduling. Rather than always simply applying the factor-of-two approximation, we can design an anytime approximation algorithm and allocate it some amount of processor time based on expectations regarding its performance. The fully-polynomial approximation scheme[4] of Ibarra and Kim [30] for solving the optimization version of the knapsack problem serves nicely as the basis for an anytime approximation algorithm for choosing method sequences. The simplest approach would be to classify the base-level problems in terms of, say, the number of goals and the length of time until the deadline, and gather statistics on the utility derived from invoking the approximation scheme with different precision requirements. Whether or not this more complicated approach to deliberation scheduling performs better than the factor-of-two approximation will depend upon the specifics of the application and how efficiently the algorithms are realized. It is easy to imagine applications, however, for which the expected performance of the system will be improved by using such a scheme.

The use of flexible computations can also simplify problems in which one decision procedure produces an intermediate result that is used as input to a second decision procedure. Boddy and Dean [4] investigate one such problem involving a robot courier assigned the task of delivering packages to a set of locations. The robot has to determine both the order in which to visit the locations, referred to as a *tour*, and, given a tour, plan paths to traverse in moving between consecutive locations in the tour. To simplify the analysis, it is assumed that the robot's only concern is time: it seeks to

[4] See Garey and Johnson [20] for a discussion of fully-polynomial approximation schemes for NP-complete problems. For our purposes, an approximation scheme S for a problem Π takes an instance I_Π and a precision requirement $\epsilon > 0$ and returns a candidate solution that is within ϵ of the optimal solution. Such a scheme is said to be fully polynomial just in case the time complexity of S is bounded by a polynomial function of $\frac{1}{\epsilon}$ and the size of I_Π.

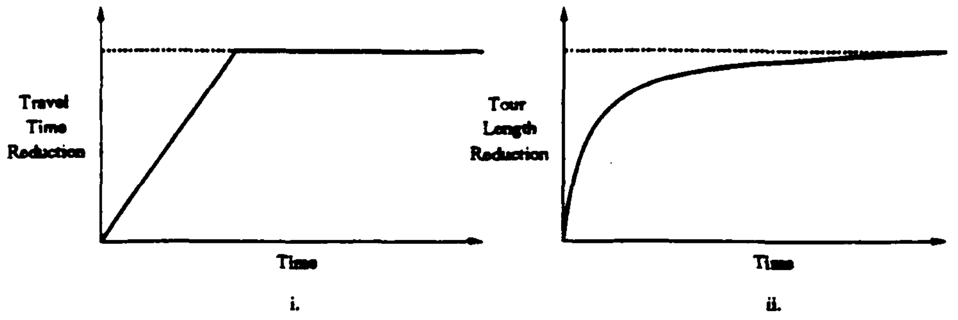


Figure 8.3: Performance profiles relating (i) the expected savings in travel time to time spent in path planning, and (ii) the expected reduction in the length of a returned tour as a function of time spent in tour improvement.

minimize the total amount of time consumed both in sitting idly deliberating about what to do next, and in actually moving about the environment on its errands. Furthermore, it is assumed that there is no advantage to the robot in starting off in some direction until it knows the first location to be visited on its tour. Finally, while the robot can deliberate about any subsequent paths while traversing a path between consecutive locations in the tour, it must complete the planning for a given path before starting to traverse that path.

The two primary components of the decision making process involve generating the tour and planning the paths between consecutive locations in the tour. The first is referred to as *tour improvement* and the second as *path planning.* Boddy and Dean employ iterative refinement approximation routines for solving each of these problems, and gather statistics on their performance to be used at run-time in guiding deliberation scheduling. The statistics are summarized in what are called *performance profiles.* Figure 8.3 (from [4]) shows the profiles for path planning and tour improvement. Figure 8.3.i shows how the expected savings in travel time increase as a function of time spent in path planning. Figure 8.3.ii shows how the expected length of the tour decreases as a fraction of the shortest tour for a given amount of time spent in tour improvement. In the analysis described in [4], this performance estimate is independent of the initially selected tour. We assume that the robot starts out with an initial randomly selected tour. Given the length of some initial tour, the expected reduction in length as a function of time spent in tour improvement, and some assumptions about the performance of path planning, the robot can figure out exactly how much time to devote to tour improvement in order to minimize the time spent in stationary deliberation and combined deliberation and traversal.

There currently is no general theory about how to combine anytime algorithms, and neither is there likely to be in the near future. For cases in which the decision problems are not independent, there is not a great deal that we can say. However, for the case of independent decision problems for which anytime decision procedures exist, or for which a pipelined sequence of anytime decision procedures exist, as in the robot courier problem, there is a great deal of interesting research to be done: research that can draw heavily on the scheduling and combinatorial optimization literature.

even this
generality

It is worth pointing out some connections between the Russell and Wefald work and that of Dean and Boddy and Horvitz. The Russell and Wefald work can be seen as trying to construct an optimal anytime algorithm: a single algorithm that operates by calculating a situation-specific estimate of utility using only local information, just as subscribed by information value theory. It should be possible to apply the Russell and Wefald approach to scheduling anytime algorithms. For some purposes (*e.g.*, the game-playing and search applications described in [41]), the monolithic approach of Russell and Wefald seems perfectly suited. For other applications (*e.g.*, the robotic applications described in [4] or the intensive-care applications described in [28]), it is quite convenient to think in terms of scheduling existing approximation algorithms.

Since this book is concerned with planning and control problems, we now turn our attention to the general class of time-dependent planning problems mentioned earlier, and investigate the deliberation scheduling issues that arise with regard to various subclasses of this general class.

8.4 Time-Dependent Planning

We define a class of time-dependent problems in terms of

1. A set of event (or *condition*) types, $\mathcal{C}$
2. A set of action (or *response*) types, $\mathcal{A}$
3. A set of time points, $\mathcal{T}$
4. A set of decision procedures, $\mathcal{D}$

RC doesn't fit the model, really: time(c) is not well-defined

5. A value function, V

We assume that at each point in time the agent knows about some set of pending events that it has to formulate a response to. We are not concerned with how the agent came to know this information; suffice it to say that the agent has some advance r otice of their type and time of occurrence. To represent its knowledge regarding future events, we say that the agent knows about a set of *tokens* drawn from the set $\mathcal{C} \times \mathcal{T}$. When we talk about events or conditions, we will be referring to tokens and not types. Each condition, c, has a type, type(c) $\in \mathcal{C}$, and a time of occurrence, time(c) $\in \mathcal{T}$. In the following, all conditions are assumed to be instantaneous (*i.e.*, corresponding to point events).

We evaluate the agent's performance entirely on the basis of its responses. Let Response(c) $\in \mathcal{A}$ be the agent's response to the condition c. Let V($a|c$) be the value of responding to the condition c with the action $a \in \mathcal{A}$. To simplify the analysis, we make the strong assumption that the value of the agent's response to one condition is completely independent of the value of the agent's response to any other condition. Given this independence assumption, we can determine the total value of the agent's response to a set of conditions C as the sum,

$$\sum_{c \in C} \mathrm{V}(\mathrm{Response}(c)|c).$$

Since we are primarily interested in investigating issues concerning the costs and benefits of computation, we abstract the problem somewhat more in order to simplify the analysis. We require the agent to formulate a response to every condition it is confronted with. We further require that the agent perform all of its deliberations regarding a given event prior to the time of occurrence of that event. There is no benefit to be had in coming up with a response early.

For each condition type, c $\in \mathcal{C}$, there is a decision procedure in dp(c) $\in \mathcal{D}$. The agent knows how to select an appropriate decision procedure given the type of an event. The decision procedures in $\mathcal{D}$ have the properties of flexible computations that we discussed earlier. In addition, the agent has expectations about the performance of these decision procedures in the form of performance profiles. For each condition type, $c \in \mathcal{C}$, there is a corresponding function $\mu_c : \mathbf{R} - \mathbf{R}$ that takes an amount of time, δ, and returns the expected value of the response to c generated by dp(c) having been run for the specified amount of time.

$$\mu_c(\delta) = \mathrm{E}(\mathrm{V}(\mathrm{Response}(c)|c, \mathrm{alloc}(\delta, \mathrm{dp}(c)))).$$

In the following, we consider various restricted classes of decision procedures. We begin by considering decision procedures whose performance profiles can be represented or suitably approximated by piecewise linear monotonic increasing functions. We add the further restriction that the slopes of consecutive line segments be decreasing. If the functions representing the performance profiles were everywhere differentiable, this restriction would correspond to the first derivative function being monotonic decreasing.[5]

Let $C = \{c_1, \ldots, c_n\}$ be the set of conditions that the system has to formulate responses for, and $\hat{t}$ be the present time. Let μ_i be the function describing the performance profile for the decision procedure used to compute responses for the ith condition. We present an algorithm that works backward from the time of occurrence of the last event in C. On every iteration through the main loop, the program allocates some interval of time to deliberating about its response to one of the conditions, c, whose time of occurrence, time(c), lies forward of some particular time t. The set of all conditions whose time of occurrence lies forward of some particular time t is denoted as

$$\Lambda(t) = \{c|(c \in C) \wedge (\mathrm{time}(c) \geq t)\}.$$

The criterion for choosing which decision procedure to allocate processor time to is based on the expected gain in value for those decision procedures associated with the conditions in $\Lambda(t)$. The criterion also has to account for the time already allocated to decision procedures. Let $\gamma_i(x)$ be the slope of the linear segment of μ_i at x unless μ_i is discontinuous at x in which case $\gamma_i(x)$ is the slope of the linear segment on the positive side of x. We refer to $\gamma_i(x)$ as the *gain* of the ith decision procedure having already been allocated x amount of processor time.

Having allocated all of the time forward of t in previous iterations, figuring out how much time to allocate on the next iteration is a bit tricky. It is certainly bounded by $t - \hat{t}$; we cannot make use of time that is already

[5] In [13], we suggested a similar restriction referred to as *diminishing returns*, and defined as follows: $\forall c \in C. \exists f. \mu_c(t) = f(t)$ such that f is monotonic increasing, continuous, and piecewise differentiable. $\forall x, y \in \mathbf{R}^+$, such that $f'(x)$ and $f'(y)$ exist, $(x < y) \supset (f'(y) \leq f'(x))$. For this class of problems, we provided an approximation algorithm that uses time-slicing to come within ϵ of optimal. The algorithm presented here is exact given the restrictions on the form of performance profiles.

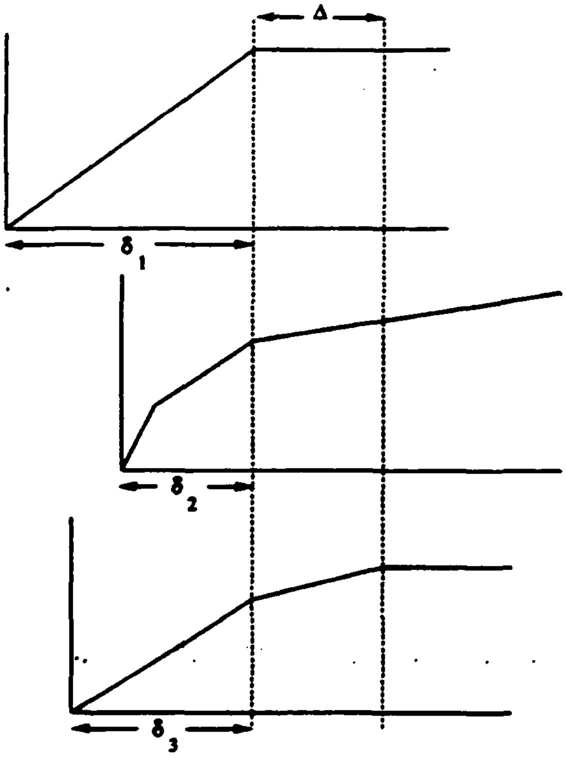


Figure 8.4: Determining min_alloc($\{\delta_1, \delta_2, \delta_3\}$)

This delay can be solved out at the Δ.

So can [illegible] $\propto 1-e^{-\lambda t}$

Probably these are just [illegible] time,

```
Procedure DS
    ;; Initialize the $\delta_i$'s to 0.
    for $i$ = 1 to $n$,
        $\delta_i \leftarrow 0$
    ;; Set $t$ to be the time of the last event in $C$.
    $t \leftarrow \mathrm{last}(+\infty)$
    ;; Allocate time working backward from the last event.
    until $t = \hat{t}$,
        ;; Compute the amount of time to allocate next.
        $\Delta \leftarrow \min\{t-\hat{t}, t-\mathrm{last}(t), \mathrm{min\_alloc}(\{\delta_i\})\}$
        ;; Find the procedure index with the maximum gain.
        $i \leftarrow \arg_i \max\{\gamma_i(\delta_i) | c_i \in \Lambda(t)\}$
        ;; Allocate the time to the appropriate procedure.
        $\delta_i \leftarrow \delta_i + \Delta$
        ;; Decrement time by the amount of allocated time.
        $t \leftarrow t - \Delta$
    ;; Set $t$ to be the current time.
    $t \leftarrow \hat{t}$
    ;; Schedule working forwards in continuous segments.
    for $i$ = 1 to $n$,
        ;; Assume that $\mathrm{time}(c_i) \leq \mathrm{time}(c_j)$ for all $i < j$.
        run the $i$th decision procedure from $t$ til $t + \delta_i$
        ;; Increment time by the amount of allocated time.
        $t \leftarrow t + \delta_i$
```

Figure 8.5: Deliberation scheduling procedure

past. In addition, given that we are using the gain of the decision procedures for conditions in $\Lambda(t)$ as part of our allocation criterion, the criterion only applies over intervals in which $\Lambda(t)$ is unchanged. Let last(t) be the first time prior to t that a condition in C occurs that is not already accounted for in $\Lambda(t)$:

$$\mathrm{last}(t) = \max\{\mathrm{time}(c) | c \in C - \Lambda(t)\}$$

Finally, given that the gains determine the slope of particular line segments characterizing the performance of the decision procedures, we have to be careful not to apply our criterion to an interval longer than that over which the current gains are constant. Let min_alloc($\{\delta_i\}$) be the minimum of the lengths of the intervals of time for the next linear segments for the performance profiles given the time allocated thus far. Figure 8.4 illustrates min_alloc($\{\delta_i\}$) for a particular case.

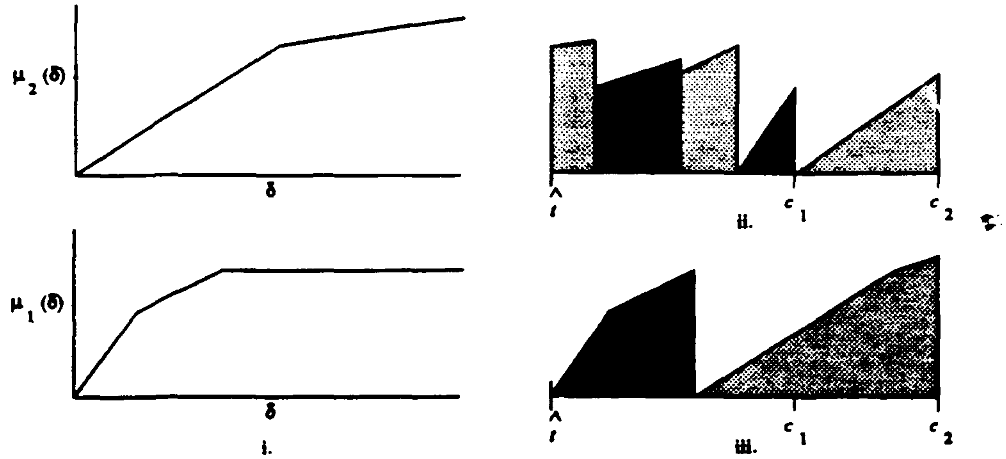


Figure 8.6: A simple example of deliberation scheduling

Figure 8.5 lists the procedure for deliberation scheduling for the class of problems under consideration. The procedure, DS, consists of three iterative loops. The first initializes the allocation variables, the second determines how much time to allocate to each of the decision procedures, and the third determines when the decision procedures will be run. For convenience, we assume that the events in C are sorted so that time$(c_i) \leq$ time(c_j) for all $i < j$. This assumption is only made use of in determining when decision procedures will be run.

Consider the following simple example to illustrate how DS works. Suppose that we have two events to contend with, c_1 and c_2. Figure 8.6.i shows the performance profiles for the decision procedures for c_1 and c_2. DS starts by allocating all of the time between c_1 and c_2 to the decision procedure for c_2. The next interval of time to be allocated (Δ) is determined by the first linear segment of μ_1, and this interval is allocated to c_1.

At this point, the slope of the second linear segment of μ_1 is less than the slope of the first segment of μ_2, so the next interval (determined by what is left of the first linear segment of μ_2) is allocated to c_2. The next interval corresponds to the second linear segment of μ_1, and this entire interval is allocated to c_1. Finally, the remainder of μ_1 has slope 0, so the remaining time is allocated to c_2. Figure 8.6.ii shows the complete history of allocations, and Figure 8.6.iii shows how the decision procedures are scheduled to run. Now we prove that DS is optimal.

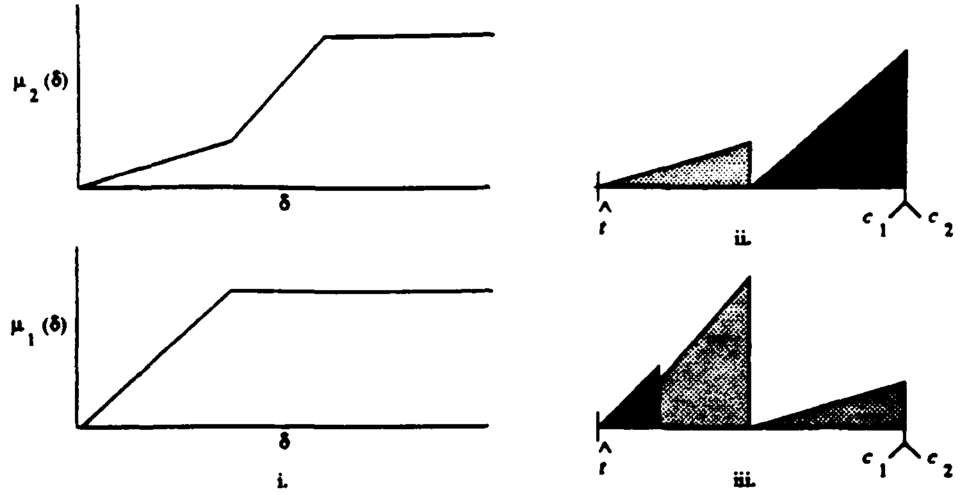


Figure 8.7: Performance profiles that foil DS

Theorem 1 *The procedure DS is optimal in the sense that it generates a set of allocations $\{\delta_i\}$ maximizing $\sum_i^n \mu_i(\delta_i)$.*

Proof: We proceed by induction on n, the number of conditions in C. For the basis step, $n = 1$, the algorithm allocates all of the time available to the only event in C, and hence is clearly optimal. For the induction step, we assume that DS is optimal for up to and including $n - 1$ events. Our strategy is to prove each of the following:

1. Let $\hat{t}'$ be the time of the earliest event in C. Using $\hat{t}'$ as the starting time, DS optimally allocates processor time to the $n - 1$ (or fewer assuming simultaneously occurring events) events in C occurring after $\hat{t}'$.

2. DS optimally allocates processor time to all n events in C over the period from $\hat{t}$ until the time of occurrence of the first event in C, accounting for the processor time already committed to in the allocations described in Step 1.

3. Given Steps 1 and 2, the combined allocations result in optimal allocations for C starting at $\hat{t}$.

Step 1 follows immediately from the induction premise. To prove Step 2, we have to demonstrate that DS solves the simpler problem of maximizing

$\sum_i^n \mu_i(\delta_i)$ subject to the constraint that $\sum_i^n \delta_i = l$, where l is the length of time separating $\hat{t}$ and the first event in C. For this demonstration, it is enough to note that, as long as the set of events being considered $(\Lambda(t))$ remains unchanged, during each iteration of the main loop, DS chooses an interval with maximal gain, and, by making this choice, DS in no way restricts its future choices given that all subsequent intervals are bound to have the same or smaller gains. This last point is due to the restriction that the slopes of consecutive line segments for all performance profiles are decreasing. Note that, if we were to relax this restriction, the greedy strategy used by DS would not produce optimal allocations. Figure 8.7.i provides a pair of performance profiles such that DS will produce suboptimal allocations. Figure 8.7.ii shows the allocations made by DS, and Figure 8.7.iii shows the optimal allocations.

Step 3 follows from the observation that the allocation of the time from the occurrence of the first event to the occurrence of the last is independent of any consideration of the first event or any time available for deliberation prior to the occurrence of the first event. □

Theorem 1 proves that the allocations made by DS are optimal in a well-defined sense. We still have to show that the method for scheduling when to run decision procedures is correct. In particular, we have to show that DS generates a *legal* schedule, where a legal schedule is one such that for all $c \in C$ the time allocated to the decision procedure for c is scheduled prior to the time at which c occurs. To see that DS does generate legal schedules, note that DS ensures that the sum of the time allocated to all conditions that occur prior to t for any $t > \hat{t}$, is less than $t - \hat{t}$. DS is guaranteed to generate a legal schedule since it schedules all of the time for any condition c before any condition occurring later.

In the time-dependent planning problems described above, the exact time of occurrence of conditions is known by the deliberation scheduler. One can easily imagine variants in which the scheduler only has probabilistic information about the time of occurrence of events.

For instance, for each condition, c_i, the scheduler might possess a probability density function,

$$p_i(t) = \Pr(\mathbf{occurs}(t, c_i)),$$

indicating the probability that a particular condition will occur at time, t. For practical reasons, we will assume that for each condition, c_i, there is a

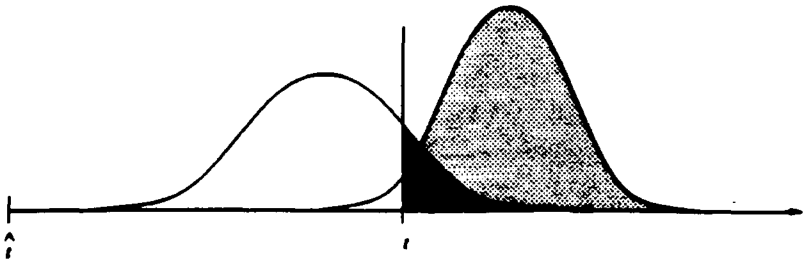


Figure 8.8: Uncertainty about the occurrence of conditions

latest time, $\sup(c_i)$, and an earliest time, $\inf(c_i)$, such that

$$\rho_i(\sup(c_i)) = \rho_i(\inf(c_i)) = 0.$$

While the scheduler does not know exactly when conditions will occur, we assume that the executor will know when to carry out a given action. For instance, conditions might have precursor events signaling their immediate occurrence. The executor would simply take the best response available at the time the precursor event for a given condition is observed.

Our performance criterion for *deliberation scheduling* is no longer,

$$\sum_{c \in C} \mathrm{V}(\mathbf{Response}(c)|c)),$$

but rather,

$$\sum_{c \in C} \int_{\hat{t}}^{\infty} \rho(\mathbf{occurs}(t,c))\mathrm{V}(\mathbf{Response}(c,t)|c)\,dt,$$

where $\mathbf{Response}(c,t)$ indicates the response generated with respect to condition, c, given that c occurs at t.

In deciding how to allocate an interval of processor time given uncertainty about the occurrence of conditions, we have to account for the possibility that the event may have already occurred. Figure 8.8 depicts the probability density functions for the time of occurrence of two conditions. The areas of the shaded regions indicate the probability that the conditions occur in the future of the time marked t.

We extend the δ_i notation to represent processor schedules. Let each δ_i be a function,

$$\delta_i : \mathbf{R} \rightarrow \{0,1\},$$

```
Procedure DS'($\Delta$)
    ;; Initialize the $\delta_i$'s to 0.
    for $i$ = 1 to $n$,
        $\delta_i \leftarrow 0$
    ;; Set $t$ to the latest possible time of occurrence.
    $t \leftarrow \max\{\inf(c_i)|c_i \in C\}$
  until $t \leq \hat{t}$
      ;; Find the index with maximum expected gain.
      $i \leftarrow \arg_i \max\{E(V(\Delta|\delta_{i|t}))|c_i \in C\}$
      ;; Allocate the time to the appropriate procedure.
      $\delta_i \leftarrow \delta_i + \langle \min\{\Delta, t - \hat{t}\}, t\rangle$
      ;; Decrement time by the amount of allocated time.
      $t \leftarrow t - \min\{\Delta, t - \hat{t}\}$
```

Figure 8.9: Deliberation scheduling with uncertain condition times

where $\delta_i(t) = 1$ if the decision procedure for c_i is allocated the processor at t, and $\delta_i(t) = 0$ otherwise. The expected value of a given schedule, δ_i, beginning at t, and allocating processor time to deliberating about a condition, c_i, is just the sum over all times, t', in the future of t, of the probability that c_i occurs at t multiplied by the expected value of the response generated by the decision procedure for c_i given the processor time scheduled between t and t'. We notate this expected value,

$$E(V(\delta_{i|t})) = \int_t^\infty \rho_i(\tau)\mu_i(\delta_i(t,\tau))\,d\tau.$$

where $\delta_i(t,t')$ is the total amount of time allocated to c_i by the schedule δ_i between t and t'. The expected value of augmenting a given schedule, δ_i, starting at t, by allocating the time from $t-\Delta$ to t to deliberating about c_i is defined by

$$\mathbf{E}(\mathbf{V}(\Delta|\delta_{i|t})) = \int_{t-\Delta}^\infty \rho_i(\tau)\mu_i(\delta_i(t,\tau,\Delta))\,d\tau - \int_t^\infty \rho_i(\tau)\mu_i(\delta_i(t,\tau))\,d\tau.$$

where $\delta_i(t,\tau,\Delta)$ is the total amount of time between t and t' allocated to c_i by the Δ-augmented schedule.

Figure 8.9 lists a procedure for deliberation scheduling for the class of problems involving uncertainty in condition times. The procedure listed in Figure 8.9 takes a positive real number, $\Delta \in \mathbf{R}^+$, to be used as the length of the interval of time allocated in each iteration of the main loop of the procedure. The assignment, $\delta_i \leftarrow 0$, results in $\delta_i(t) = 0$ for all $t \in \mathbf{R}$. The assignment, $\delta_i \leftarrow \delta_i + \langle t, t' \rangle$, results in $\delta_i(\tau) = 1$ for all τ in the interval $\langle t, t' \rangle$, and for all τ outside the interval $\langle t, t' \rangle$ is the same as it was prior to the assignment. In the following, we make several comments regarding DS'.

The first comment concerns what exactly it is that DS' computes. DS' provides an approximation to the optimal deliberation schedule. It is an approximation because we allocate each interval of length Δon the basis of expectations computed for a single point at the boundary of that interval: in general, this method will result in a suboptimal deliberation schedule. On the positive side, the smaller the allocated intervals are, the better the approximation; the schedules generated by DS' converge to the optimal schedules as $\Delta \rightarrow 0$. On the negative side, the smaller Δ is, the longer it takes to compute the entire deliberation schedule.

In this chapter, we generally ignore the cost of deliberation scheduling, assuming that, if the running time of the scheduling algorithm is linear in the size of the input, then the cost of scheduling is negligible. In this case, however, the cost of deliberation scheduling can be made arbitrarily large by employing a small enough value for Δ. In practice, it will be necessary to account for the cost of deliberation scheduling. In some cases, it will reasonable to choose a value for Δ at *compile time* by *experimenting* with various values and expected inputs. In other cases, it might be useful to select a value at run time, using some simple criteria for selection; this constitutes a simple example of meta-meta-reasoning.

The second comment regarding DS' concerns the form of the final schedule. Unlike the case in which we know exactly when each condition will occur, we cannot coalesce all of the time allocated to a given condition into a continuous interval. As a consequence, we have to assume the capability of switching the processor rapidly between different decision procedures. In most multi-tasking operating systems, assuming this sort of rapid process switching is reasonable.

The final comment regarding DS' concerns the notion of optimality which we employ in rating performance. Claims of optimality are made assuming that there will be no further opportunities to modify the schedule. In practice, however, each time that a condition occurs, it will be useful to compute a new deliberation schedule.

In the remainder of this section, we consider one more variant of time-dependent planning. In this variant, we assume that there are no external conditions requiring responses of the controller; instead, the controller has some number of tasks it is assigned to carry out. The tasks do not have to be completed by any particular time, but the sooner they are completed the better. As in the previous problems, we assume that there is a decision procedure for each task. Generally, the more time the controller deliberates about a given task, the less time it takes to carry out that task. We assume that the outcome of deliberation concerning one task is independent of the outcome of deliberation concerning any other.

This does not fit - [illegible] formalism ?

The performance profiles relate the time spent in deliberation to the time saved in execution. For instance, suppose that the task is to navigate from one location to another, and the decision procedure is to plan a path to follow between the two locations; up to a certain limit, the more time spent in path planning, the less time spent in navigation.

In the following, we consider a few special instances of this class of problems. In the first instance, all of the deliberations are performed in advance of carrying out any task. This model might be appropriate in the case in which a set of instructions are compiled in advance, and then loaded into a robot that carries out the instructions. Deliberation scheduling is simple. For each task, the scheduler allocates time to deliberation as long as the time spent in deliberation results in a greater reduction in the time spent in execution. All of the deliberation is then performed in advance of any execution.

In the second instance, the order in which the tasks are to be carried out is fixed in advance, and all deliberation concerning a given task is performed in advance of carrying out that task, but deliberation concerning one task can be performed while carrying out another. In this instance, deliberation scheduling is somewhat more complicated. We consider deliberation scheduling in terms of three steps, *minimal allocation*, *dead-time reduction*, and *free-time optimization*. In the minimal allocation step, we proceed as in the previous instance, by determining a minimal allocation for each task, ignoring the possibility of performing additional deliberation during execution.

This minimal allocation for a given task corresponds to that allocation of deliberation time minimizing the sum of deliberation and expected execution time. Figure 8.10.i shows four tasks and the time they are expected to take, assuming no time spent in deliberation. Figure 8.10.ii shows the performance profiles for each of the four tasks. The dotted line in each performance profile indicates the minimum slope such that allocating deliberation time will result in a net decrease in the sum of deliberation and expected execution time for the minimal allocation. Figure 8.10.iii shows the minimal allocations for each of the four tasks, where the δ_i indicates the

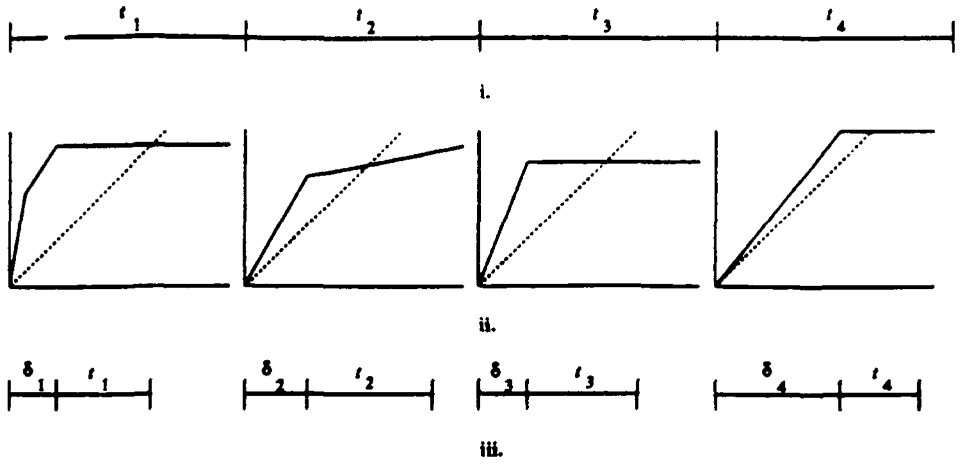
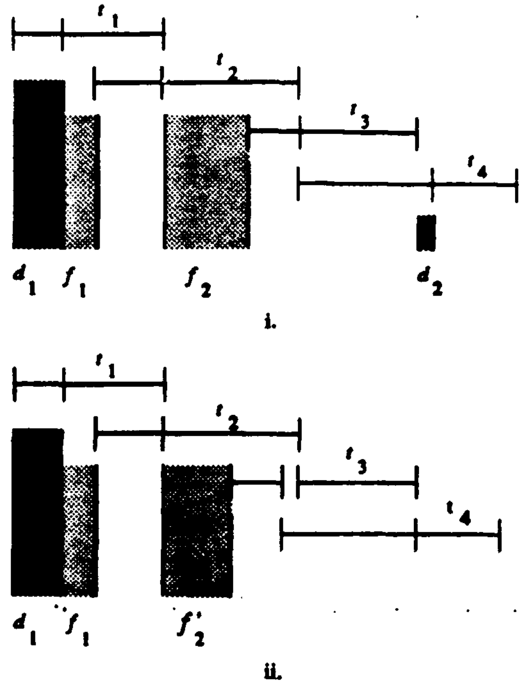


Figure 8.10: Minimal allocations of processor time

time allocated to deliberation for t_i.

Using the allocations computed in the minimal allocation step, we construct a schedule in which tasks begin as early as possible subject to the constraint that all of the deliberation for a given task occurs in a continuous block immediately preceding the task and following any deliberation for the previous task. Figure 8.11.i shows the resulting schedule for the example of Figure 8.10. Note that there are two additional types of intervals labeled in Figure 8.11.i. This first type, notated f_i, indicates the *free time* associated with t_i, corresponding to time when the system is performing a task but not deliberating. The second type, notated d_i, indicates the *dead time* associated with t_i, corresponding to time when the system is deliberating but not performing any task.

In the dead-time reduction step, we attempt to reduce the amount of dead time in the minimal-allocations schedule by making use of earlier free time. Where possible, we allocate earlier free time to performing the deliberations previously performed during the dead time, starting with latest dead time intervals and working backward from the end of the schedule and using the latest possible intervals of free time. Figure 8.11.ii shows the schedule of Figure 8.11.i modified to eliminate one of the dead time intervals. It is not always possible to eliminate all dead time intervals. In particular, any deliberation time allocated for the first task will always correspond to dead time.



i.

ii.

Figure 8.11: Reducing dead time using available free time

Following this ,rocess of dead-time reduction. if there is any free time left. we attempt to allocate it for deliberating about other tasks. This is just a bit tricky. since by performing additional deliberation we eliminate previously available free time. Not only do we eliminate the free time we are filling in by scheduling deliberation. but the deliberation reduces execution time thereby eliminating additional free time. There is one special case for which optimally allocating the additional free time is easy. This is the case in which all of the performance profiles are piecewise linear composed of two linear segments such that slope of the first segment is the same for all profiles and the slope of the second is 0. This corresponds to the specification of the robot courrier problem described in the previous section, regarding the task of optimally allocating processor time for planning several paths between locations in a tour of such locations to be visited.

The reason that optimally allocating the additional free time in this case is easy is explained as follows. If the slope of the first linear segment for all of the performance profiles is greater than 1, then all of the time corresponding nonzero slope will be allocated in making the minimal allocations, and any additional allocations will yield no decrease in execution time. If the slope of the first linear segment for all of the performance profiles is less than 1. then all of the minimal allocations will be 0, and there will be no free time to allocate.

I [illegible]
[illegible]

There are many variations on the problems described above. This section is meant as a sampler of problems and associated deliberation scheduling techniques. Deliberation scheduling should be seen as a means of programming in knowledge about how to improve run-time performance. There are occasions. however, in which the time required to apply that knowledge is not available at run time, and it becomes reasonable to make certain choices concerning the allocation of computational resources at design time. In the next section. we consider design-time tradeoffs for improving system performance.

8.5 Compiling Problem Solving Systems

In the previous sections, we were concerned with the design of systems that. given expectations about the performance of decision-making routines, were able to make appropriate tradeoffs at run-time so as to maximize expected utility. Another approach to building systems capable of good performance in time-critical situations involves making certain inferences at design time

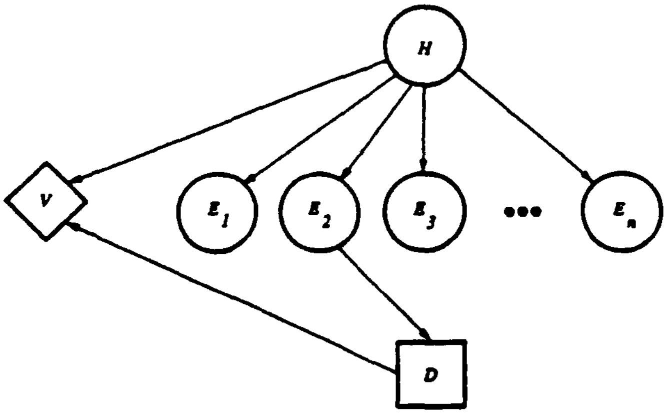


Figure 8.12: Decision model for the diagnosis problem (after [26])

and caching those inferences for use at run time in order to improve the system's response time. Other researchers have suggested compiling domain models to guarantee bounded response time [31, 39]. Generally, the result of compilation is a table or circuit whose *space requirements* are an important factor in assessing the value of a given compilation method. Usually, the object is to improve response time without sacrificing decision quality; when this cannot be done (*e.g.*, the storage requirements for caching are substantial) it becomes necessary to consider tradeoffs. The approaches described in this section are noteworthy for their use of a decision theoretic criterion for trading space for response time.

Heckerman, Breese, and Horvitz [26] investigate a simple form of tradeoff that involves improving response time by compiling decision models. In their model, the utility of a state depends on whether or not a particular hypothesis H is true and whether or not an action D is taken. We will assume that, if H is true, the action D should be taken, and otherwise the action $\neg D$ is appropriate. We can define a threshold probability of H, call it p^*, such that the agent is indifferent about acting one way or the other:

$$p^* U(H, D) + (1 - p^*) U(\neg H, D) = p^* U(H, \neg D) + (1 - p^*) U(\neg H, \neg D).$$

The agent is not able to observe H directly, and, hence, must infer whether or not H is true on the basis of the observed evidence, $E_1, E_2, \ldots, E_n$. Thus the agent should perform the action D if and only if

$$\Pr(H | E_1, E_2, \ldots, E_n) > p^*.$$

The resulting decision mode' (depicted graphically in Figure 8.12) is represented as an *influence diagram* that captures the causal and informational dependencies between chance variables (indicated as circles) and between chance variables and decision variables (indicated as boxes), and the value of states of the world corresponding to particular instantiations of the chance and decision variables (indicated as diamonds).

Heckerman, Breese, and Horvitz reformulate the decision problem in terms of log-likelihood ratios, and, by making certain independence assumptions, they reduce the decision problem to computing

$$W = \sum_{i=1}^{n} w_i$$

where the w_i are the weights accorded to the E_i. The agent should perform the action D if and only if $W > W^*$ where W^* is the log-likelihood equivalent of p^*. We will refer to the strategy of computing the weights at run time as the *compute* strategy.

As an alternative to computing the sum of the weights of evidence at run time, the agent might consider all possible combinations of evidence and compile a table indicating whether or not to act for each possible combination. If memory is inexpensive and response time critical, then this might be an attractive alternative. In general, however, it will be prohibitively expensive to compile a table for all possible combinations of the evidence, and, hence, if the agent wants to speed its response time by compiling a table, it will have to limit its attention to a subset of the evidence. Suppose that the agent chooses m pieces of evidence,

$$\{E_{c_1}, E_{c_2}, \ldots, E_{c_m}\} \subseteq \{E_1, E_2, \ldots, E_n\},$$

to use in compiling a table of responses. For each of the 2^m possible combinations of the m variables, we compute the sum,

$$W_m = \sum_{i=1}^{m} w_{c_i},$$

at compile time, and store D in the table if $W_m > W^*$ and $\neg D$ otherwise. At run time, the agent simply uses the evidence as an index to lookup the appropriate entry in the table. We will refer to a strategy of compiling a table for m pieces of evidence as a *compile* strategy.

Note that the advantage of the compute strategy is that it takes all of the evidence into account; the disadvantage is that there may be some delay

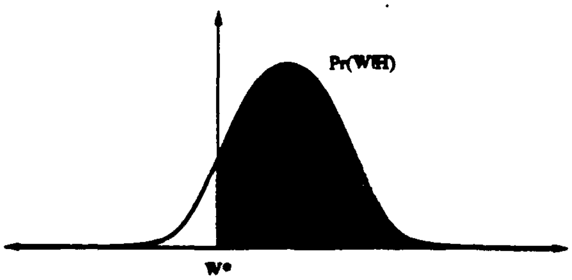


Figure 8.13: The probability that the total evidential weight will exceed the threshold is determined by summing the area under the curve for the distribution of W given H and above the threshold weight W_m (after [26]).

between the time that the evidence is observed and the time that the agent responds to the evidence. The compile strategy may enable the agent to respond more quickly, but at the cost of ignoring some of the evidence and providing storage for a decision table whose size is exponential in the number of pieces of evidence accounted for in the reduced model.

In the following, let EV_{compute} indicate the expected value of the agent using the compute strategy for a single instance of the decision problem, PC^{H}_{compute} and $PC^{\neg H}_{\text{compute}}$ indicate the cost due to computing delays in the case in which H is true and the case in which H is not, and MC_{compute} indicate the one time cost of memory for the compute strategy. Assume similar quantities for the compile strategy. In order to compare the compute strategy against different compile strategies (*i.e.*, compilation involving different subsets of $\{E_1, E_2, \ldots, E_n\}$), Heckerman, *et al.* introduce formulae for determining the *net inferential value* of a given strategy.

$$NIV_{\text{compute}} = \rho[EV_{\text{compute}} - \Pr(H)PC^{H}_{\text{compute}} - \Pr(\neg H)PC^{\neg H}_{\text{compute}}] - MC_{\text{compute}}$$

$$NIV_{\text{compile}^m} = \rho[EV_{\text{compile}^m} - \Pr(H)PC^{H}_{\text{compile}^m} - \Pr(\neg H)PC^{\neg H}_{\text{compile}^m}] - MC_{\text{compile}^m}$$

where the NIV_{compile^m} depends upon the particular choice of evidence, and ρ is a factor "that converts the expected value of each policy on a single instance to a summary (present) value for a series of problem instances over the life of the system." Given the above, the agent designer should choose the compute strategy over the compile strategy if and only if

$$NIV_{\text{compute}} > NIV_{\text{compile}^m}.$$

In the analysis presented in [26]. $PC^{\neg H}_{\text{compute}}$, PC^{H}_{compute}, and MC_{compute} are linear functions of n, the total number of evidence variables in the complete model. $PC^{\neg H}_{\text{compile}^m}$ and $PC^{H}_{\text{compile}^m}$ are linear functions of m, the total number of evidence variables in the restricted compilation model, and MC_{compile^m} is a linear function of 2^m. The formulae for the expected value of using the compute and compile strategies for a single instance of the decision problem are given as follows:

$$\begin{aligned}&EV_{\text{compute}} = \\ &\quad Pr(W > W^*|H)\mathrm{U}(H,D) + \Pr(W \leq W^*|H)\mathrm{U}(H,\neg D)]\Pr(H) + \\ &\quad [\Pr(W > W^*|\neg H)\mathrm{U}(\neg H,D) + \Pr(W \leq W^*|\neg H)\mathrm{U}(\neg H,\neg D)]\Pr(\neg H)\end{aligned}$$

$$\begin{aligned}&EV_{\text{compile}^m} = \\ &\quad [\Pr(W_m > W^*|H)\mathrm{U}(H,D) + \Pr(W_m \leq W^*|H)\mathrm{U}(H,\neg D)]\Pr(H) + \\ &\quad [\Pr(W_m > W^*|\neg H)\mathrm{U}(\neg H,D) + \Pr(W_m \leq W^*|\neg H)\mathrm{U}(\neg H,\neg D)]\Pr(\neg H)\end{aligned}$$

The only trick to using the above to decide whether to use the compute or compile strategy is determining the probabilities involving the weights (*e.g.*, $\Pr(W > W^*|H)$). Assuming that n is large (as it should be for us to take seriously the cost of computing W), then we can compute the first two moments for the each of the weights given H and combine them to approximate the distribution of W given H using the central limit theorem. Using the resulting approximations for $\Pr(W_m|H)$, $\Pr(W_m|\neg H)$, $\Pr(W|H)$, and $\Pr(W|\neg H)$, we can determine the values for the terms needed to compute EV_{compute} and EV_{compile} (see Figure 8.13).

Heckerman, *et al.* go on to consider relaxing certain assumptions (specifically, allowing multiple-valued hypothesis, evidence, and decision variables and introducing alternatives to caching complete tables, in the form of caching situation/action rules in asymmetrical trees), and methods for considering what subsets of the set of all evidence variables to consider for compilation. What they don't consider, and what might be worth pursuing, are mixed strategies involving some amount of design-time compilation and some amount of run-time inference.

If the basic methods described by Heckerman, *et al.* for evaluating the expected performance of decision models used for time-critical applications

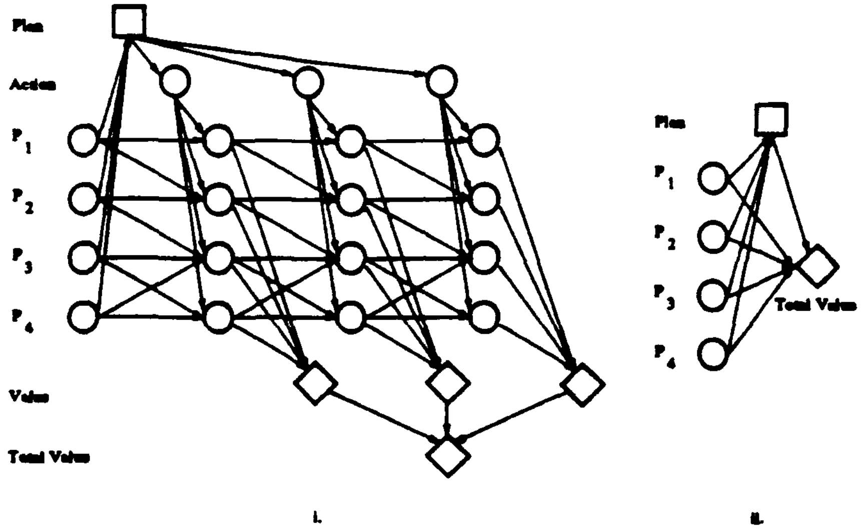


Figure 8.14: Two influence diagrams indicating (i) a complete decision model for reasoning about plans, and (ii) a reduced version of the decision model obtained by absorbing chance nodes in the complete model.

turn out to be practical for realistic decision problems, then we will want to try out more sophisticated models for reasoning about plans and change over time. Kanazawa and Dean [32] describe a model for reasoning about time, causation, and action that can be cast as an influence diagram. Given a set $\mathcal{P}$ of propositions and a set $\mathcal{T}$ of time points, we can define a set of chance variables from $\mathcal{P} \times \mathcal{T}$ representing the truth of various propositions at different points in time. By quantifying the dependencies between these chance variables, we can specify a model of change over time referred to as a *temporal Bayes net* [14].

Refer to Ch 7

The model described in [32] generalizes on this basic model of change over time to include actions so as to provide a decision model for selecting plans. Figure 8.14.i shows an example of such a model depicted as an influence diagram. Each row, except those corresponding to decision variables or value functions, indicates a proposition or quantity that changes over time, and each column indicates a different point in time. Kanazawa and Dean consider possible tradeoffs involved in improving the performance of reasoning systems using such a model for decision making. In particular, they consider trading accuracy for time by employing approximation schemes for evaluating probabilistic models [8, 28]. They also consider trading space for time by eliminating chance variables in the decision model using a method of conditioning called *node absorption* [42]. By eliminating chance variables at design time, it is possible to dramatically improve the time required to evaluate the model. Such improvements occur for both exact and approximate evaluation techniques. Figure 8.14.ii shows a version of the model shown in Figure 8.14.i obtained by repeated use of node absorption.

In general, node absorption can result in an increase in the space required to store the model: there will be fewer nodes in the resulting graph, but the space required to store the conditional probabilities quantifying the dependencies may increase significantly. However, given the structure of temporal Bayes nets, the net increase in space is generally acceptable and more than offset by the resulting reduction in evaluation time. It would be interesting to extend the techniques of Heckerman, *et al.* to evaluate at design time various alternative approximation schemes and methods of simplifying the decision model. The biggest barriers to making such extensions practical will likely be due to the combinatorics of action selection and the difficulties involved in obtaining an accurate model of the environment in the first place.

8.6 Directions for Future Research

This chapter provides only a sketch of current work on problem solving methods for time-critical applications. There is a great deal of excellent research that we did not cover, simply because it did not fit into the structure of the presentation. In particular, we did not say anything significant about architectures for real-time control [1, 7], or relate how the search community is beginning to address real-time issues [24, 33, 44]. Regarding search, Hansson and Mayer's work [24] predicts that we will find many of the standard techniques in heuristic search as emergent properties of mechanisms that employ Bayesian inference and decision-theoretic control of inference. All of this work is serving to shape a new field of research.

The next few years will see a marked increase in the effort directed at time-critical problem solving and resource-limited reasoning. We need to extend the current approaches to handle computational models that reflect the complexity of existing problem-solving systems. For instance, how might an agent deal with multiple tasks, perhaps deciding to act with regard to one task while continuing to deliberate about others. We need experience with real applications so that the research will be driven by real issues and not artifacts of our mathematical models. We need to reconcile the goal-oriented, resource-bounded perspective of artificial intelligence with the idealized, optimizing perspective taken in the decision sciences.

This chapter makes use of Howard's information value theory as a basis from which to start in analyzing systems with limited computational resources. All of the approaches described in this chapter can be seen as extensions of the basic idea of assessing the value of information sources. The approaches surveyed here depart from information value theory when they attempt to account for the cost of inference, including the computational cost of assessing the value of information sources. It would seem that the theory of experimental design [19, 35] which is concerned with the problem of maximizing the information gained from performing experiments under cost constraints might provide a source of additional techniques that could be applied in controlling inference for time-critical applications.

All of the approaches described in chapter make rather restrictive assumptions in order to avoid the combinatorics involved in dealing with unlimited decision-making horizons and complicated interactions between information sources. For practical problems, it is unlikely that we will be able to entirely relax the one-step horizon and no-competition assumptions that characterize myopic decision policies. An interesting area for future research involves identifying and dealing with restricted types of interactions and providing a disciplined approach to extending decision-making horizons. It would also be useful to explore methods of extending the anytime algorithm approaches to handle more situation-specific information.

The research on compiling decision models is just beginning, and one area that appears particularly interesting to investigate involves mixed strategies for combining design-time compilation and run-time inference. Another area that was not covered in this survey, but is of considerable interest involves learning control knowledge in the form of statistics to support decision-theoretic control of inference. Two of the papers covered in this chapter [41, 18] describe interesting techniques that address learning issues.

More about learning in general and speedup learning in particular.

The work in time-critical problem solving will have far reaching implications for the whole research community. Time is, after all, an issue in any problem solving task. Theoretical results concerning agents with limited computational resources should shed light on a number of basic representation issues. For instance, the notion of a "plan" as a persistent belief does not make sense until you take computational considerations into account.

Plans enable a system to amortize the cost of deliberation over an inte·val of time. If time were not an issue. there would be no justification in committing to a plan. What are the tradeoffs involved in generating a partial plan? What are the costs and benefits of compiling a detailed plan to use in a situation in which there will be very little time for computing appropriate responses. These are just a few of the questions that can be addressed once we begin to account for the time spent in problem solving.

8.7 Further Reading

Meta-reasoning [10. 11. 15. 16. 21. 45].

Speedup learning [34. 36].

Early work in the decision sciences on the costs and benefits of inference [17. 29. 38].

Examples of myopic decision making [3. 12].

Bibliography

[1] Agogino. A. M.. Srinivas. S.. and Schneider. K., Multiple Sensor Expert System for Diagnostic Reasoning, Monitoring, and Control of Mechanical Systems, *Mechanical Systems and Signal Processing*, (1988).

[2] Barnett. V.. *Comparative Statistical Inference*. (John Wiley and Sons. New York. 1982).

[3] Ben-Basat, M.. Myopic Policies in Sequential Classification. *IEEE Transactions on Computers*, **27** (1978) 170–174.

[4] Boddy, Mark and Dean, Thomas, Solving Time-Dependent Planning Problems, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI. 1989. 979–984.

[5] Bodin. L. and Golden, B., Classification in Vehicle Routing and Scheduling. *Networks*. **11** (1981) 97–108.

[6] Brachman, Ronald J.. Levesque, Hector J., and Reiter, Raymond, (Eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, (Morgan-Kaufmann, Los Altos, California, 1989).

[7] Breese, John S. and Fehling, Michael R., Decision-Theoretic Control of Problem Solving: Principles and Architecture, *Proceedings of the 1988 Workshop on Uncertainty in Artificial Intelligence. Minneapolis, MN*, 1988, 30–37.

[8] Chavez. R. Martin. *Fully Polynomial Randomized Approximation Schemes for the Bayesian Infererencing Problem*. Report KSL-88-72. Section on Medical Informatics. Stanford University School of Medicine. 1988.

[9] Chernoff, Herman and Moses, Lincoln E., *Elementary Decision Theory*, (John Wiley and Sons, New York, 1959).

[10] Davis, Randall, Teiresias: Applications of Meta-Level Knowledge, Davis, Randall and Lenat, Douglas B., (Eds.), *Knowledge-Based Systems in Artificial Intelligence*, (McGraw-Hill International Book Company, 1982), 227–490.

[11] de Kleer, Johan, Doyle, Jon, Steele Jr., Guy L., and Sussman, Gerald Jay, AMORD: Explicit Control of Reasoning, Brachman, Ronald J. and Levesque, Hector J., (Eds.), *Readings in Knowledge Representation*, (Morgan-Kaufmann, Los Altos, CA, 1985), chapter 19, 346–355. Originally published in 1977.

[12] de Kleer, Johan and Williams, Brian C., Diagnosing Multiple Faults, *Artificial Intelligence*, **32**(1) (1987) 97–130.

[13] Dean, Thomas and Boddy, Mark, An Analysis of Time-Dependent Planning, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 49–54.

[14] Dean, Thomas and Kanazawa, Keiji, A Model for Reasoning About Persistence and Causation, *Computational Intelligence*, **5**(3) (1989) 142–150.

[15] Doyle, Jon, *A Model for Deliberation, Action, and Introspection*, Technical Report AI-TR-581, MIT AI Laboratory, 1980.

[16] Doyle, Jon, Reasoning, Representation, and Rational Self-Government, Ras, Zbigniew W., (Ed.), *Methodologies for Intelligent Systems, 4*, New York, North-Holland, 1989; 367–380.

[17] Edwards, Ward, Dynamic Decision Theory and Probabilistic Information Processing, *Human Factors*, **4** (1962) 59–73.

[18] Etzioni, Oren, Tractable Decision-Analytic Control, In Brachman et al. [6], 114–125.

[19] Fedorov, V., *Theory of Optimal Experimental Design*, (Academic Press, New York, 1972).

[20] Garey, Michael R. and Johnson, David S., *Computing and Intractibility: A Guide to the Theory of NP-Completeness*, (W. H. Freeman and Company, New York, 1979).

[21] Genesereth, Michael R., An Overview of Metalevel Architecture, *Proceedings AAAI-83, Washington, D.C.*, AAAI, 1983, 119–123.

[22] Good, I. J., A Five Year Plan for Automatic Chess, *Machine Intelligence*, 4 (1962) 59–73.

[23] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G., Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Proceedings Discrete Optimization, Vancouver*, 1977.

[24] Hansson, Othar and Mayer, Andrew, The Optimality of Satisficing Solutions, *Proceedings of the 1988 Workshop on Uncertainty in Artificial Intelligence, Minneapolis, MN*, 1988, 148–157.

[25] Hayes-Roth, Barbara, Washington, Richard, Hewett, Rattikorn, Hewett, Michael, and Seiver, Adam, Intelligent Monitoring and Control, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 243–249.

[26] Heckerman, David E., Breese, John S., and Horvitz, Eric J., The Compilation of Decision Models, *UW89, Windsor, Ontario*, 1989, 162–173.

[27] Horvitz, Eric J., Reasoning About Beliefs and Actions Under Computational Resource Constraints, *Proceedings of the 1987 Workshop on Uncertainty in Artificial Intelligence, Seattle, Washington*, 1987.

[28] Horvitz, Eric J., Suermondt, H. Jacques, and Cooper, Gregory F., *Bounded Conditioning: Flexible Inference for Decisions Under Scarce Resources*, Technical Report KSL-89-42, Stanford Knowledge Systems Laboratory, 1989.

[29] Howard, Ronald A., Information Value Theory, *IEEE Transactions on Systems Science and Cybernetics*, 2(1) (1966) 22–26.

[30] Ibarra, O. H. and Kim, C. E., Fast Approximation Algorithms for the Knapsack and Sum of the Subset Problems, *Journal of the ACM*, 22 (1975) 463–468.

[31] Kaelbling, Leslie Pack, Goals as Parallel Program Specifications, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 60–65.

[32] Kanazawa, Keiji and Dean, Thomas, A Model for Projection and Action, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 985–990.

[33] Korf. Richard. Real-Time Heuristic Search: New Results. *Proceedings AAAI-88. St. Paul. Minnesota.* AAAI. 1988. 139–144.

[34] Laird. J. E., Newell. A., and Rosenbloom. P. S., SOAR: An Architecture for General Intelligence. *Artificial Intelligence.* **33** (1987) 1–64.

[35] Mendenhall. W., *Introduction to Linear Models and the Design and Analysis of Experiments.* (Wadsworth. Belmont. California. 1968).

[36] Minton. S., Carbonell. J. G., Knoblock. C. A., Kuokka. D. R., Etzioni. O., and Gil. Y., Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence.* **40** (1989) 63–118.

[37] Pearl. Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* (Morgan-Kaufmann. Los Altos. California. 1988).

[38] Raiffa. Howard and Schlaifer. R., *Applied Statistical Decision Theory,* (Harvard University Press. 1961).

[39] Rosenschein. Stan. Synthesizing Information-Tracking Automata from Environment Descriptions. In Brachman et al. [6], 386–393.

[40] Russell, Stuart J. and Wefald, Eric H., On Optimal Game-Tree Search using Rational Meta-Reasoning, *Proceedings IJCAI 11. Detroit. Michigan.* IJCAI. 1989. 334–340.

[41] Russell, Stuart J. and Wefald. Eric H., Principles of Metareasoning, In Brachman et al. [6].

[42] Shachter. Ross D., Evaluating Influence Diagrams. *Operations Research.* **34**(6) (1986) 871–882.

[43] Simon, Herbert A. and Kadane. Joseph B., Optimal Problem-Solving Search: All-or-None Solutions. *Artificial Intelligence.* **6** (1975) 235–247.

[44] Smith. David E., *A Decision-Theoretic Approach to the Control of Planning Search.* Report No. LOGIC-87-11. Stanford Logic Group. 1988.

[45] Weyhrauch. R. W., Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence.* **13** (1980) 133–170.

Chapter 9

Learning in Planning and Control

In the problems considered in previous chapters, we are given a model of the physical process we are trying to control and a specific ~~to~~ goal to achieve or performance index to maximize. The model provided may not be the most accurate model possible, but once given there is no attempt made to improve upon it. In order to choose appropriate actions to take, the controller has to predict the consequences of its actions as those consequences relate to the goal or performance index provided in the problem specification. In this chapter, we consider problems in which the system can use its experience, the perceived record of its interaction with the environment, to improve upon its performance by improving its ability to predict the consequences of its actions.

The concept of learning, as it is used in everyday speech, is difficult define precisely. Intuitively, learning has something to do with changing behavior in response to experience. However, if we were to equate learning with changing behavior in response to experience, we would be obliged to say that using sensor data to determine what action to take next was a form of learning. Rather than debate what is and what is not learning, we simply coopt the word for our own purposes and equate it with certain forms of *function approximation*.

In the simplest form of function approximation for control, we assume that some aspect of the environment can be modeled by a particular function. We generally assume that this function does not change over time, or,

[0] ©1990 Thomas Dean. All rights reserved.

if it does change, it changes very slowly. The control system is given examples in the form of inputs to the function and their corresponding outputs. From these examples, the system is supposed to find an approximation to the function of interest that agrees on the examples seen so far and generalizes to those that it has not seen as yet. This type of learning is called *supervised learning* since the control system is told exactly what is expected for each input provided during learning.

We talk about approximations instead of exact functions for a number of reasons. By specifying in advance a parameterized family of functions to represent the function of interest, we can often simplify the search involved in finding a candidate function. The parameterized family of functions also allows us to limit the amount of storage used to represent the function of interest. One drawback to the use of a restricted family oi functions is that the function of interest may not belong to the specified family and so we must choose the function that best approximates the function of interest. A second reason for using approximations is that the control system has to continually respond to its environment, and, at any given point in time, it will want to use whatever information it has so far to guide its choice of action.

What constitutes a good approximation will depend on any number of factors relating to the performance of the controller. For instance, the amount of storage required to represent the function, the amount of time required to evaluate the function for a given input, and how the results of evaluating the function impact on the ability of the controller to achieve its goal or maximize its performance index are all factors that have to be taken into account in evaluating a given approximation.

In previous chapters, we represented control problems and their solutions using a variety of functions. For instance, the evolution of the state of a dynamical system was represented as a function from states and inputs to states, and a performance index was represented as a function from sequences of states and inputs to the real numbers. A typical control scheme might involve enumerating a set of possible courses of action, predicting their consequences in terms of the state trajectories corresponding to the predicted evolution of the system state, and then comparing the various courses of action by applying a value function to the corresponding state trajectories. This is roughly the approach taken in Chapter 6 with respect to stochastic dynamic programming and in Chapter 7 on using Bayesian decision theory for planning.

In this chapter, we consider problems similar to those investigated in

Chapters 6 and 7. In particular, we model the dynamical system as a stochastic process, and we assume a separable value function in which the total value of a state trajectory is the (temporally discounted) sum of the value (reward) at each state. The big difference between the problems of this chapter and those of the earlier chapters is that the controller will not be given the state-transition probabilities for the dynamical system nor will it be given the immediate reward function.

There are two basic approaches to building a controller for problems in which the dynamics and rewards are not initially specified. In the first approach, the controller attempts to learn the dynamics and rewards, and then constructs an optimal policy for the resulting model as in Chapters 6 and 7. We call this approach the *explicit-model* approach. In the second approach, the controller attempts to learn an optimal policy by constructing an evaluation function to use in selecting the best action to take when in a given state. The controller constructs this evaluation function without recourse to an explicit model of the system dynamics, and so, while the system cannot predict what the state resulting from a given action will be, it can determine whether that resulting state is better or worse than the state resulting from any other action. We call the second approach the *direct* approach.

In the explicit-model approach, the control system has to learn two functions. First, it has to learn the dynamics, a function from states and actions to distributions over states. Second, the system has to learn a function from states and actions to the real numbers. From these two functions, the system constructs a third function, a policy or control law, from states to actions.

Of course, it is not as simple as, learn the dynamics and rewards, and then construct a policy and follow it ever after. The control system has to continue to operate while it is learning the dynamics and rewards, and this introduces some complications reminiscent of the interaction between observation and control in systems for which the separation property does not hold. The problem is that the controller has to visit all of the states and try out all of its options in every state sufficiently often to construct an accurate statistical model. This means that the controller has to systematically explore its environment and experiment with various policies in order to ensure that it will construct an optimal policy.

In the direct approach, the system also learns two functions. First, it learns a function from states to the real numbers. This function is essentially the value function for a fixed policy introduced in Chapter 6, but here we attempt to learn this function without the use of an explicit dynamical model. Second, the system learns a function from states and actions to the real numbers that is used for selecting what action to take next. Here again the problem of exploration and experimentation comes up. The calculation of the value function assumes a fixed policy, but the controller has to deviate from the fixed policy in order to explore its environment in sufficient detail to find the optimal policy.

In both the explicit model and direct approaches, the ultimate objective is to learn an optimal policy, a function from states to actions that maximizes expected cumulative discounted reward. The system does not, however, learn by being given examples of states and the optimal actions to take in those states. Rather, the system performs actions in states and is given feedback in the form of rewards. This type of learning is called *reinforcement* learning.

Reinforcement learning is complicated by the fact that the reinforcement in the form of rewards is often intermittent and delayed. The controller may perform a long sequence of actions before receiving any reward. This makes it difficult to attribute credit or blame to actions when a reward finally is received. In chess or checkers, reinforcement occurs in the form of lost pieces or lost games, and the reason for losing a piece or a game is seldom completely due to the last action taken before the loss. The problem of attributing credit or blame in such circumstances is called the *credit-assignment* problem, and any solution to the problems addressed in this chapter will require a solution to the credit-assignment problem.

The rest of this chapter is organized as follows. First, we consider some basic techniques for learning functions. We then return to the problem of learning an optimal policy, concentrating on the direct approach described above. In looking at the problem of learning an optimal policy, a number of computational issues become critical in considering problems with large input spaces. We consider approaches that address the problem of coping with large input spaces. We then take another look at learning optimal policies in terms of learning rules. Finally, we consider some issues concerned with the ability of a learning system to perceive the true state of the world.

9.1 Function Approximation

We characterize a function-learning problem in terms of

- a domain set X,

- a range set Y, and
- a set of candidate functions $F = \{f : X \rightarrow Y\}$.

In the cases we are interested in, the domain is often the state or output space of a dynamic system, and the range is often the input space of a dynamic system or the real numbers in the case of learning a value function. In most cases, the set of candidate functions can characterized by a finite set of parameters.

For instance, in the case in which $X = Y = \mathbf{R}$, the set,

$$\left\{C_0 + C_1x + C_2x^2 + C_3x^3 | C_0, C_1, C_2, C_3 \in \mathbf{R}\right\},$$

represents the set of all polynomials of degree 3 or less, and is characterized by four real-valued parameters.

The size of the parameter set is often a good indication of the storage required for a given function learning problem. In some cases, the storage required for a problem is equal to the size of the domain set. For instance, suppose that the domain set is a finite subset of the integers, $X \subset \mathbf{Z}$, and the range is the real numbers. Consider the set of candidate functions,

$$\left\{\sum_{i \in X} C_i \mathcal{I}_i(x) | C_i \in \mathbf{R}\right\},$$

where $\mathcal{I}_i$ is the characteristic or *indicator* function for the singleton set consisting of just i and defined by

$$\mathcal{I}_i(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{if } x \neq i \end{cases},$$

In this case, we have one real-valued parameter for each element of X.

It may be difficult, impossible or even unnecessary to characterize the set of candidate functions using a finite set of parameters. It may be difficult or impossible if the function varies erratically or randomly over some portion of its domain. It may unnecessary if all we require is an approximation of the function. For the problems we are interested in, a good approximation will suffice for acceptable control. For instance, in learning a value function for control, all the controller cares about is whether performing one action is better than performing another; being able to compute an exact value or even a value to 10 significant digits is not likely to improve the performance of the controller.

Let X be any set, $\{X_i | 1 \leq i \leq n\}$ partition[1] X, and $Y = \mathbf{R}$. Consider the set of candidate functions,

$$\left\{ \sum_{i=1}^{n} C_i \mathcal{I}_i(x) | C_i \in \mathbf{R} \right\},$$

where, in this case, $\mathcal{I}_i$ is the indicator function for the set X_i,

$$\mathcal{I}_i(x) = \begin{cases} 1 & \text{if } x \in X_i \\ 0 & \text{if } x \notin X_i \end{cases}.$$

In this case, we have partitioned the domain into a finite set of regions and assigned a single real-valued parameter to each region. This allows us to represent exactly a class of piecewise-constant functions with n pieces where the pieces correspond to the regions of the partition. We can approximately represent a much larger class of functions.

You can probably think of several, more general methods of characterizing classes of candidate functions. For instance, the set of regions need not define a partition; the regions might intersect or the set might not cover the entire domain. In addition, the set of regions need not remain static thoughout the learning process; their boundaries might be characterized by additional parameters.

The regions referred to above are often called *receptive fields* in the literature on artificial neural networks. In some cases, each receptive field is characterized by two parameters, a point in the domain set, $\mathbf{R}^n$, and a diameter, together describing an n-dimensional spherical region of the domain. Each receptive field has associated with it a small amount of storage used to represent some aspect of the behavior of the function in the region covered by the field. These fields can be moved about to obtain a better approximation of the function. Large fields can be used to represent the behavior of the function in regions where not much is going on. Several small fields can be used to represent the behavior of the function in regions where a lot is going on.

In addition to allowing the regions to vary, the behavior of the function in a given region can be characterised by any finitely parameterizable function. The variety of learning problems is considerable, and it is not our purpose here to survey those problems in any detail. In the following, we consider

[1] Let $\{X_i\} = \{X_1, X_2, \ldots, X_n\}$. We say that $\{X_i\}$ *partitions* X just in case, $X_i \subset X$ for $1 \leq i \leq n$, $\bigcup_{i=1}^{n} X_i = X$, and $X_i \cap X_j = \emptyset$ for all i and j such that $i \neq j$.

a very restricted sort of function learning in order illustrate some basic principles and provide some machinery that will be of use in subsequent sections.

In the following, we assume that the range set is the real numbers, and consider only very simple sets of candidate functions of the form,

$$F = \left\{ \sum_{i=1}^{n} w_i \phi_i(x) \,|\, w_i, \phi_i(x) \in \mathbf{R} \right\},$$

where $\phi_i : X \rightarrow \mathbf{R}$ is an arbitrary function, and we use the notation, w_i, for the parameters to indicate that they are variable *weights*.

The set of functions, $\{\phi_i\}$, are often called *features* in the literature. Such features might model measurements taken by different sensors that detect whether or not a specific property holds of the input, x. In general, each function, ϕ_i, processes the input in some manner and issues a real number which is weighted by the parameter, w_i, and combined with the other features. The functions so represented are linear combinations of the features though the features themselves need not be linear functions.

We can rewrite

$$\left\{ \sum_{i=1}^{n} w_i \phi_i(x) \,|\, w_i, \phi_i(x) \in \mathbf{R} \right\},$$

in vector notation as

$$\{\mathbf{w}\boldsymbol{\phi}(x) | \mathbf{w}, \boldsymbol{\phi}(x) \in \mathbf{R}^n\},$$

where the first term, called the *parameter vector*, is defined by

$$\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle,$$

the second term, called the *feature vector*, is defined by

$$\boldsymbol{\phi}(x) = \langle \phi_1(x), \phi_2(x), \ldots, \phi_n(x) \rangle,$$

and the implied operator separating the two vectors is the inner product.

To indicate a member of F, it is enough to specify a vector $\mathbf{w} \in \mathbf{R}^n$. Learning generally proceeds by incrementally adjusting the weights to specify an updated parameter vector. At any given point, the learning system will have seen a set of input/output pairs,

$$\{\langle x_i, y(x_i) \rangle | 1 \leq i \leq k\}.$$

where $y(x)$ denotes the output of the function we are trying to learn for the input, x. One standard criterion for selecting weights is to determine the parameter vector that minimizes the mean of the squared error. That is, we wish to find $\mathbf{w} \in \mathbf{R}^n$ minimizing the sum,

$$\frac{1}{k}\sum_{i=1}^{k}\epsilon(x_i)^2,$$

where the *error* term, $\epsilon(x)$, is defined as

$$\epsilon(x) = y(x) - \mathbf{w}\phi(x).$$

If we are willing to keep around the entire sequence of input/ouput pairs, we could compute the parameter vector minimizing the mean of the squared error directly. The mean of the squared error is a convex function of the weights and hence it has a unique minimum. As a consequence, we can compute the parameter vector minimizing the sum of the squared error by simply setting the gradient,

$$\nabla\left[\frac{1}{k}\sum_{i=1}^{k}\epsilon(x_i)^2\right] = -\frac{2}{k}\sum_{i=1}^{k}\epsilon(x_i)\phi(x_i),$$

to zero and solving the resulting system of equations for the weights. Alternatively, we can use gradient-descent search methods to find the weights. Recall that gradient-descent search proceeds by making small changes to the parameter vector in the direction indicated by the negative gradient.

It is generally assumed, however, that either the system cannot afford the storage to keep around all of the training data, or that it would be useless to keep around all of the training data given that the function we are trying to learn changes gradually over time. In keeping with this assumption, we are interested in methods that proceed by making small changes to the parameter vector on the basis of the last example.

Let $\mathbf{w}_t$ and x_t denote, respectively, the parameter vector and the example at time t. In a manner similar to that employed in gradient descent, we make adjustments to the parameter vector on the basis of the last example, using the following update rule,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \beta\epsilon_t(x_t)\phi(x_t),$$

where the error term in this case is defined as

$$\epsilon_t(x) = y(x) - \mathbf{w}_t\phi(x).$$

and the scalar, β, is the learning rate or *gain* of the update rule. This update rule is called the *least mean square* (LMS) rule and is due to Widrow and Hoff [21]. This rule is also closely related to the *perceptron* learning rule of Rosenblatt developed for pattern classification [15].

If there is storage available, we can improve the estimate of the gradient by taking into account more than just the last example. Generalizing on the LMS rule, we have the rule,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \beta\frac{1}{k}\sum_{i=t-k-1}^{t} \epsilon_t(x_i)\boldsymbol{\phi}(x_i),$$

accounting for the last k examples.

In order for the above learning method to converge to a fixed parameter vector closely approximating the function of interest, the sequence of training examples has to represent a sufficiently varied subset of the set of all such examples. Exactly what constitutes a sufficiently varied set of examples will depend upon the class of functions being learned, but, intuitively, you want examples drawn from across the domain with more examples in regions where the behavior of the function is more complex.

Experiment 1 To illustrate the performance of the function-learning approach described above, suppose that the target function is the cubic polynomial,

$$y(x) = 1.20 - 0.2x + 3.1x^2 - 0.9x^3,$$

and the examples are drawn (pseudo) randomly from the set,

$$\{(x, y(x)) | -1 \leq x \leq 1\}.$$

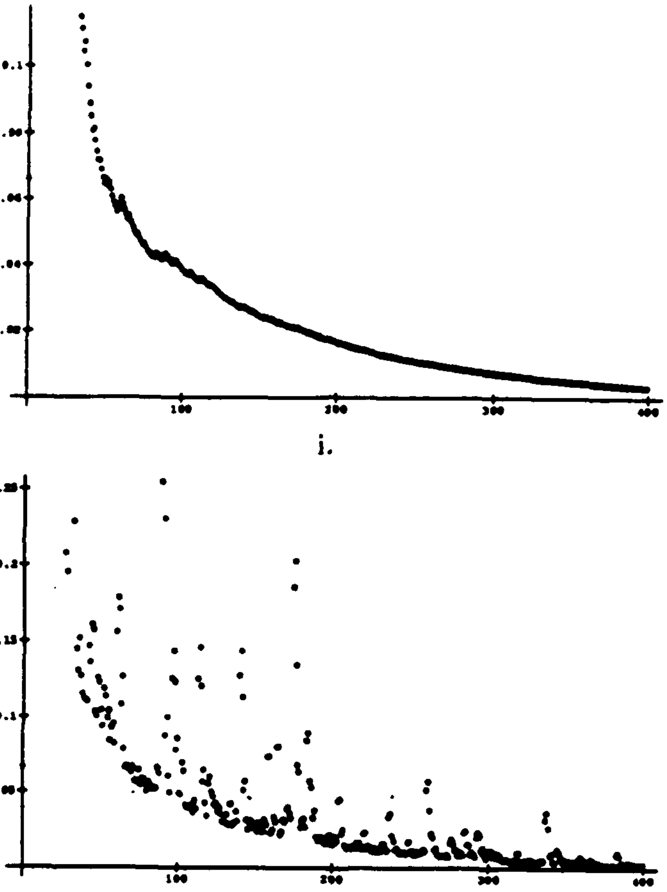
Figure 9.1.i shows the performance of the LMS update rule with $k = \infty$ (*i.e.*, use all of the examples encountered so far) and $\beta = 0.1$.[2] The approximation after 400 examples is

$$\mathbf{w}\boldsymbol{\phi}(x) = 1.311442 - 0.290810x + 2.900718x^2 - 0.763296x^3.$$

Figure 9.1.ii $k = 1$ shows the performance of the LMS update rule with $k = 1$ (*i.e.*, use only the last example encountered) and $\beta = 0.1$. The approximation after 400 examples is

$$\mathbf{w}\boldsymbol{\phi}(x) = 1.292668 - 0.238687x + 2.966245x^2 - 0.81559x^3.$$

[2] Hideki Isozaki supplied the data for the graphs shown in Figure 9.1.



i.

ii.

Figure 9.1: Performance of the generalized LMS rule

Now we have techniques that will allow us to select a good approximation from a set of candidate functions given a set of training examples. We can utilize any *a priori* knowledge we have of the function of interest to bias the learning process by selecting appropriate features to constrain the set of candidate functions. In selecting a set of features to represent the problem. one can make the learning problem trivial (*e.g.*. you select the function of interest as one of the features) or impossible (*e.g.*, the function of interest cannot be closely approximated by a linear combination of the features).

The performance of a function approximation technique is measured in terms of the amount of storage required. the time required for each update. and the expected accurracy of the approximation (*e.g.*, the mean squared error) as a function of the number of training examples seen so far. There are a host of other function approximation techniques. but their performance invariably depends upon starting with a good representation. The linear method utilizing the LMS rule described above is probably the best understood method, and, despite its limitations (*e.g.*, it can only be used to represent functions that can be described as a linear combination of the features), it is often the method of choice in building practical learning systems.

The learning methods discussed in this section can also be viewed as special-purpose memories. In the case of there being one parameter per member of the domain set, learning corresponds to just filling in the entries in a large table. In some cases. the set of features allow the learning system to generalize from the set of examples seen so far to those that it has yet to see. It is this notion of generalization. that people often closely associate with learning. Once again, the ability of a system to generalize depends critically upon the representation chosen.

9.2 Policy and Value Learning

As indicated in the introduction to this chapter. we intend to narrow the scope of our discussion to focus on learning an optimal policy for a stochastic sequential decision making task. We are interested in any route to the goal of learning an optimal policy, but the discussion of Chapter 6 suggests one relatively straightforward approach. The approach is to learn the transition probabilities and the reward function and then employ Howard's policy iteration technique to compute the optimal policy.

Let X be the state space of the dynamical system. and U be the input space. Assuming that it is possible to directly observe the state of the dynamical system, the controller would start by executing a random walk (*i.e.*, it would select its actions according to a uniform distribution). Let $\delta : X \times U \times X \rightarrow \mathbf{Z}$ be the *transition-statistics* function, and $\mu : X \times U \times X \rightarrow \mathbf{R} \times \mathbf{Z}$ be the *reward-statistics* function. Initially, let $\delta(x, u, x') = 0$, and $\mu(x, u, x') = \langle 0, 0 \rangle$ for all $x, x' \in X$ and $u \in U$. Every time that the controller performs an action, u, in state, x, resulting in next state, x', the controller would update the transition-statistics function by incrementing $\delta(x, u, x')$ by one. Similarly, every time the controller receives a reward, r, in state, x', having started in state, x, and performed action, u, the controller would update the reward-statistics function so that $\mu(x, u, x') = \langle s + r, n + 1 \rangle$, where prior to the update $\mu(x, u, x') = \langle s, n \rangle$. After a period of time determined by how accurate a model is required, we would compute estimates of the transition probabilities,

$$\Pr(x(t+1) = x' | x(t) = x, u(t) = u) = \frac{\delta(x, u, x')}{\sum_{x'' \in X} \delta(x, u, x'')},$$

and rewards,

$$R(x, u, x') = \frac{s}{n} \text{ where } \mu(x, u, x') = \langle s, n \rangle,$$

and use policy iteration to compute the optimal policy given the estimates for the rewards and transition probabilities.

In theory, the approach outlined above is perfectly reasonable. There are, however, disadvantages. First, it may not be desirable for a robot to perform a random walk during the training period; the robot might become a nuisance or damage itself. Second, the transition probabilities may change gradually over time; a robot with a fixed training period may construct an initially optimal policy, but that policy might become significantly suboptimal as the transition probabilities change over time. Third, policy iteration is computationally rather expensive. We consider each of these three disadvantages in turn.

With regard to performing a random walk during training, the robot has to explore the space of possible state transitions thoroughly enough to obtain reliable statistics. This does not mean, however, that the robot has to perform actions that are obviously dangerous or socially incorrect, since those actions will, presumably, never be a part of an optimal policy anyway. One obvious method for avoiding dangerous or antisocial behavior is to build the learning system on top of a base controller that only exhibits safe, socially correct behavior. In this case, the outputs of the learning system are the inputs to the base controller. This basic idea of building a learning system on top of an existing controller applies to any approach to learning.

With regard to the transition probabilities changing over time, there is no need to have a fixed training period in the scheme outlined above. The controller could continually gather statistics on the rewards and transition probabilities and periodically update its policy. The only problem is that the controller may not obtain adequate statistics if it always follows what it believes to be the optimal policy. Hence, in addition to periodically updating its policy, the controller will have to periodically engage in some exploratory behavior in order to assure that its estimated rewards and transition probabilities are accurate.

The problem in dealing with computational costs is a bit more troubling. Policy iteration is polynomial in the sizes of the state and input spaces. Value determination, which is performed once in each iteration of the policy iteration procedure, requires solving a system of $|X|$ simultaneous linear equations. If most of the :ransition probabilities are not zero, simply representing this system of equations takes $O(|X|^2)$ space, but keep in mind that, in the case of mostly nonzero transition probabilities, it will require $O(|X \times U \times X|)$ space just to store the transition probabilities.

This problem arising from the sizes of the state and input spaces is often called the *curse of dimensionality*. Generally, the state and input spaces can be viewed as a cross product of subspaces. For example, we might represent the state space, X, as an n-dimensional product space,

$$X = \prod_{i=1}^{n} X_i,$$

where $\{X_1, X_2, \ldots, X_n\}$ are the component subspaces. Each subspace, X_i, might represent a different property of the environment (*e.g.*, the robot's current position, orientation, or amount of remaining fuel). Some of the component subspaces might represent a finite discretization of an infinite space.

Individually, the sizes of the subspaces might be modest, but the prospect of quantifying over a product space of size,

$$|X| = \prod_{i=1}^{n} |X_i|,$$

can be daunting from a computational perspective. This can be especially frustrating if large portions of that product space are unreachable (*e.g.*, if the robot's battery is completely dis harged it cannot have a positive velocity). or uninteresting (*e.g.*. the robot might be able to detect light. but. for most tasks. the intensity of light has no influence on the robot's choice of action as it navigates using sonar).

The curse of dimensionality raises a deep issue that will not go away: it is not problem that can be solved. In the following section. we return to this issue. but for the time being we ignore it and consider some approaches that circumvent some of the problems that arise regarding computing optimal policies.

Suppose. for the sake of argument. that the controller has a time- and storage-efficient procedure that. given a state and an action. returns a (next) state according to the distributions specified by the dynamical system. Given this prccedure. which we refer to as the *transition oracle*, and a reward function. we can now compute an optimal policy by using the following simple stochastic approximation (Monte Carlo) routine for value determination in the standard policy iteration algorithm.

Here is the stochastic value determination routine. For each $x \in X$, compute $V(x)$ as follows. Use the transition oracle to determine m state transition histories of length k,

$$\begin{array}{c} x_{1,1}, u_{1,1}, x_{1,2}, u_{1,2}, \ldots, u_{1,k-1}, x_{1,k} \\ x_{2,1}, u_{2,1}, x_{2,2}, u_{2,2}, \ldots, u_{2,k-1}, x_{2,k} \\ \vdots \\ x_{m,1}, u_{m,1}, x_{m,2}, u_{m,2}, \ldots, u_{m,k-1}, x_{m,k} \end{array}$$

where $x_{j,1} = x$ for $1 \leq j \leq m$, the actions are determined by the current policy.

$$\eta(x_{j,i}, u_{j,i}) = x_{j,i+1},$$

and the state transitions are obtained from the transition oracle. We obtain the approximate value of the state. x. given the policy. η. as

$$V(x) = \frac{1}{m}\sum_{j=1}^{m}\frac{1}{k}\sum_{i=1}^{k}\lambda^{i}R(x_{j,i}, u_{j,i}, x_{j,i+1}).$$

This approximation converges to the true value in the limit as m and k tend to infinity. If in addition to the transition oracle. we are given a time- and storage-efficient means of computing rewards. a reward oracle. then we can compute the optimal policy in a very space efficient manner by some careful programming.

Of course, the point of this oracle business is that we do indeed have such oracles, at least in a manner of speaking. The world is our oracle: the rewards and state transitions that it visits upon us are exactly the state transitions and rewards of the physical process that we attempt to capture in our dynamical models.

In the remainder of this section, we consider methods for learning optimal policies that rely upon performing experiments in the real world rather than upon explicitly modeling the dynamics and rewards. These methods emphasize storage efficiency, and, in some cases, were originally conceived of as models of learning in biological organisms. In light of the issues that arise with regard to high-dimensional state and input spaces, this focus on storage-efficient methods is likely to have important engineering consequences as well.

In the following approach, we assume that the controller has adequate storage for a value function, $V : X \to \mathbf{R}$. In addition, we assume that the controller has storage for a function to be used in computing the policy. This might just be a policy function, $\pi : X \to U$, or it might be something a bit more complicated, for instance, a function from states and actions to the real numbers providing some expectation of cumulative reward. We assume very little in the way of computation at each state transition. We begin by considering how to learn the value function for a fixed policy, starting with a very simple case.

Consider a finite-state, deterministic dynamical system with a fixed policy. We assume that every state is reachable from every other state, and proceed as we did in the previous section on function approximation. Let $X = \{1, 2, \ldots, n\}$, and $\mathbf{v} \in \mathbf{R}^n$. Since $\mathbf{v}$ changes over time, we provide a temporal index, $\mathbf{v}_t$, to distinguish between the values at different points in time. Similarly, let x_t and r_t denote, respectively, the state and the reward at time t. Let $V_t(i) = \mathbf{v}_t[i]$, where $\mathbf{v}_t[i]$ indicates the ith component of the vector $\mathbf{v}_t$. We define the vector of features,

$$\boldsymbol{\phi}(x) = \langle \phi_1(x), \phi_2(x), \ldots, \phi_n(x) \rangle,$$

where

$$\phi_i(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{if } x \neq i \end{cases}$$

Consider the following simple update rule,

$$\mathbf{v}_{t+1} = \mathbf{v}_t + [r_{t+1} - V_t(x_t)]\boldsymbol{\phi}(x_t).$$

In this case, if the system is allowed to run indefinitely, the parameter vector will converge to a fixed value given by $V(x_t) = R(x_{t+1})$.

To handle sequential decision problems of indefinite duration with discounting rate, λ, for rewards, we employ the following variation on the above rule,

$$\mathbf{v}_{t+1} = \mathbf{v}_t + [r_{t+1} + \lambda V_t(x_{t+1}) - V_t(x_t)]\phi(x_t).$$

Here also the parameter vector converges to a fixed value, but, in this case, the value is identical with that obtained using the value determination routine of Chapter 6.

The above equations should look vaguely familiar. They have the same basic form as the LMS learning rule introduced in the previous section. In the discounting case, the error term is just the difference between the current estimate of the state value, $V(x_t)$, and the revised estimate of this value, $r_{t+1} + \lambda V_t(x_{t+1})$. The above rule simplifies to just

$$V_{t+1}(x_t) = r_{t+1} + \lambda V_t(x_{t+1}).$$

The stochastic case is somewhat more complicated. We assume a completely ergodic Markov process so that every state is visited infinitely often. In this case, the revised estimate of the value of the state, $x_t = i$, should be

$$r_{t+1} + \lambda \sum_{j=1}^{n} \rho_{ij} V_t(j), \quad \forall j$$

where ρ_{ij} is the transition probability defined for the current policy. Of course, we do not have the transition probabilities so instead we simply make use of what we do have. The update rule for the stochastic case is exactly the same as the rule for the deterministic case with one variation,

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \beta[r_{t+1} + \lambda V_t(x_{t+1}) - V_t(x_t)]\phi(x_t),$$

we introduce a learning rate, $0 < \beta \leq 1$, as in the LMS learning rule. In the stochastic case, the values do not converge to the values indicated by value determination. Instead, they fluctuate about the expected values according to the most recent state transitions. The variance in these fluctuating values is bounded, and can be made arbitrarily small by an appropriate choice of β, or reduced asymptotically to zero by choosing an appropriate *schedule* for β (*e.g.*, $\beta = \frac{1}{t}$).

Note the revised value estimates in the above equations are just a special case of estimating long-term returns on the basis of some number of observed rewards. In general, we can make use of any number of observed rewards using estimates of the form,

$$V_t(x_t) = r_{t+1} + \lambda r_{t+2} + \cdots + \lambda^{n-1} r_{t+n-1} + \lambda^n V_{t+n}(x_{t+n}).$$

Estimates with more observations are generally better in that they provide more accurate estimates and speed learning, but they also require more memory and computation.

Experiment 2 Provide an example illustrating the steady-state performance of an estimation routine using the above update rule. Use the mean of the squared error as an evaluation metric and the robot-courier problem as a test case.

Now that we have a method for computing the value function for a given policy, the next step is to develop a method for improving the current policy. To that end, we introduce the idea of learning the expected value of actions. For each state, x, and action, u, we allocate memory, $W(x, u)$, for storing an estimate, called an *action value*, of the expected value of performing that action in that state. Initially all the action values are zero. The update rule uses the value function introduced in the previous paragraphs,

$$W_{t+1}(x_t, u_t) = W_t(x_t, u_t) + \alpha[r_{t+1} + \lambda V_t(x_{t+1}) - V_t(x_t)],$$

where u_t is the action taken at time t, and all other actions values, $W_{t+1}(x, u)$ such that either $x \neq x_t$ or $u \neq u_t$, remain the same. The intuition behind this rule is as follows.

Recall that V is the estimated value function with respect to a particular policy. If u_t is the action indicated by the current policy in state x_t, then the error, $[r_{t+1} + \lambda V_t(x_{t+1}) - V_t(x_t)]$, should be zero on average. On the other hand, if u_t is some action other than that recommended by the current policy, then the error will be greater than, less than, or equal to zero on average, depending on whether or not taking that action and then following the current policy thereafter results in a higher, lower, or identical expected value compared to that for the recommended action.

Note that, assuming the controller sticks to a fixed policy, the values specified by W with the exception of those corresponding to the recommendations of the fixed policy will not converge; rather, they are likely to increase or decrease without bound.

These values do, however, provide us with useful information in deciding how to improve the current policy; the relative values tell us what actions to change in the current policy in order to define an improved policy. Consider the following approach.

1. Following the current policy and updating only the value function, perform a number of steps so that the values for the current policy are good approximations of the actual value function.
2. Set $W_t(x, u) = 0$ for all $x \in X$ and $u \in U$ where t is the current time.
3. Following a random policy, and, updating only the action values, perform a number of steps so that the relative action values are in keeping with the actual expected action values with high probability.
4. Using the relative action values, choose a new policy,

   $$\eta(x) = \arg\max_u W(x, u),$$

   and set it to be the current policy.
5. Go to Step 1.

The above method directly mimics the policy iteration routine introduced in Chapter 6 using stochastic methods instead of exact methods for the value determination and policy improvement steps. One drawback is that it is likely to take a very long time to converge to an optimal policy. As an alternative to this method, researchers have tried approaches that involve running stochastic value determination and policy improvement continuously. Instead of switching back and forth between a current estimated best policy and a random policy, these approaches generally employ a stochastic policy that, on average, chooses actions from the current estimated best policy, but, according to a fixed distribution, occasionally deviates and experiments with actions other than those recommended by the current policy. It generally helps if the value function is only updated if the action selected is the same as the action recommended by the current policy.

No one has as yet proved that these alternative approaches converge to the optimal policy, though they do appear to converge in practice. However, there is one learning method that has been shown to converge in the limit. This method is also interesting because it is a stochastic variant of the value iteration approach described in Chapter 6 rather than policy iteration approach.

Recall that value iteration is a technique that uses successive approximation to compute a value function that converges in the limit to the value function for the optimal policy. The policy at each point in time is determined by the actions that maximize the current estimate for the optimal value function. Instead of learning both a value function and a set of action values, the controller learns just the action values, but, in this approach, the action value' are updated by the following learning rule.

$$W_{t+1}(x_t, u_t) = W_t(x_t, u_t) + \alpha_t(x_t, u_t)[r_{t+1} + \lambda \max_u W_t(x_{t+1}, u) - W_t(x_t)],$$

where, in order to guarantee convergence, we have to vary the learning rate, α_t, over time according to a schedule satisfying certain requirements.

Note that in order to guarantee that the procedure will find the optimal policy in the limit, it is enough to to guarantee that W converges to the optimal value function in the limit. To guarantee that W converges to the optimal value function in the limit, it is sufficient that, for each pair consisting of a state, x, and an action, u, the following statements hold.

1. The controller attempts action, u, in state, x, an unbounded number of times as $t \to \infty$.
2. The learning rate $\alpha_t(x, u)$ tends to zero as $t \to \infty$.
3. The sum $\sum_{t=0}^{n} \alpha_t(x, u)$ increases without bound as $t \to \infty$.

Actually, these are very modest requirements. The first statement just requires that the controller not permanently ignore portions of the space of states and actions. The second and third restrictions are satisfied by a learning schedule of the form $\alpha_t(x, u) = \frac{1}{t}$.

Experiment 3 Provide an example illustrating the performance of the two learning approaches described above. Once again, use the mean of the squared error with respect to the optimal value function as the performance metric and the robot-courier problem as the test example.

At this point, we can learn an optimal policy. We have a method that is guaranteed to converge in the limit and that appears to work well in practice for simple problems. The learning methods considered in this section are generally time- and space-efficient with the exception of the memory required for storing the requisite functions. Since these functions generally require $O(|X|)$ space, it it is worthwhile considering methods to reduce this storage overhead. The next section is concerned with exactly this issue.

9.3 Coping With Large Input Spaces

Let X be the domain of the function we are interested in learning. Suppose that $|X|$ is large; so large that it is impractical to allocate storage for each $x \in X$ in the case of a finite X or for each region of a reasonable finite discretization in the case of an infinite X. If the function we are trying learn has complex behavior throughout its domain and that behavior does not generalize, then we are in trouble. However, if we are only interested in the behavior of the function in certain regions of X (we assume that we do not know these regions in advance or otherwise we would simply restrict the domain), or the behavior of the function is only occasionally of sufficient complexity to warrant significant amounts of storage for its approximation, then we can, at least in certain circumstances, learn a good approximation using an amount of storage significantly less than that required by X.

The basic idea is quite simple: we employ hashing techniques to map a large space into a significantly smaller one. The smaller space is represented by a finite number of storage elements containing the parameters for the family of candidate functions. Learning proceeds by adjusting these parameters using your favorite learning rule, LMS in the cases that we consider.

The method was originally conceived of as a computational model of motor learning in the cerebellar cortex. It was discovered by James Albus [1] and David Marr [8] independently, but it is generally referred to as the CMAC approach, after the name given to it by Albus, the Cerebellar Model Articulation Controller [2].

As was mentioned, the basic idea is to map a large space onto a smaller one using hashing. As with all hashing techniques, there is always some danger of *collisions*, the results of mapping different elements of the larger space onto the same element of the smaller space. In some cases, this is a good thing (*e.g.*, when the value of the function is the same for each element of the larger space), but, in others, it degrades performance. To avoid the bad consequences of hashing, CMAC employs several mapping functions each of which maps each point in the domain into a different storage element as shown in Figure 9.2. The output of CMAC for a given element of the domain is the average of the values in the storage elements determined by all of the mapping functions. In the following, we introduce notation to describe CMAC more precisely.

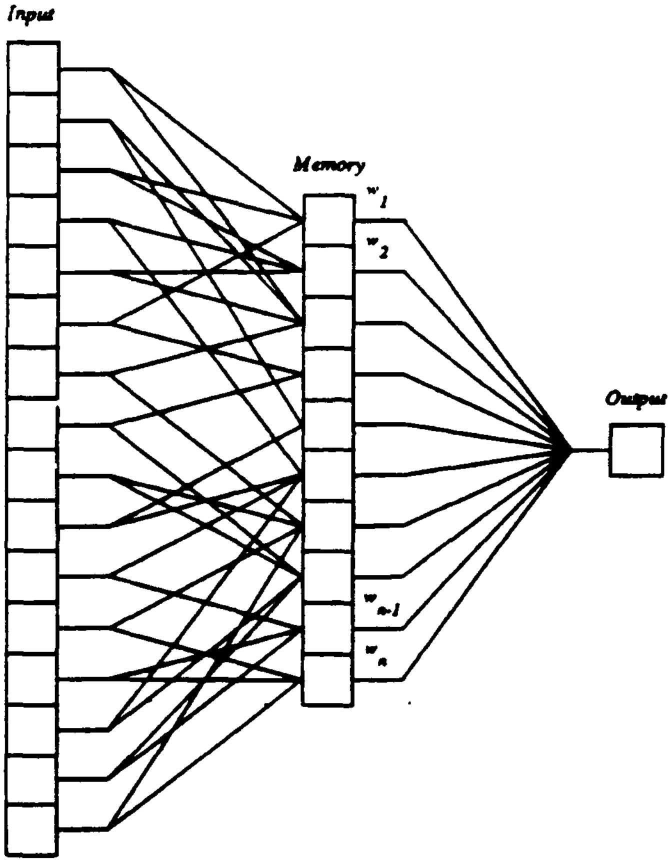


Figure 9.2: Mapping a large domain onto a smaller one

We begin by defining m partitions of the set X.

$$\begin{array}{c} X_{1,1}, X_{1,2}, X_{1,3}, \ldots \\ X_{2,1}, X_{2,2}, X_{2,3}, \ldots \\ \ldots \\ X_{m,1}, X_{m,2}, X_{m,3}, \ldots \end{array}$$

A simple and effective method of generating the m partitions for $X = \mathbf{R}^d$ is to create an initial partition, and then modify it to create the $m - 1$ remaining partitions. Each of remaining partitions is generated by uniformly displacing the regions of the initial partition by a fixed offset, so that no two partitions have the same region boundaries.

We need to define a function mapping X to the smaller set $\{1, 2, \ldots, n\}$. To provide the redundancy required to avoid the problems caused by hashing collisions, we define m functions, $\text{Map}_i : X \rightarrow \{1, 2, \ldots, n\}, 1 \leq i \leq m$, one for each of the m partitions. The ith mapping function is defined,

$$\text{Map}_i(x) = \text{Hash}(\text{Region}_i(x)),$$

where $\text{Hash} : \mathbf{Z} \rightarrow \{1, 2, \ldots, n\}$ is the hashing function, and $\text{Region}_i : X \rightarrow \mathbf{Z}$ is defined as

$$\text{Region}_i(x) = j \text{ such that } x \in X_{i,j}.$$

In the case of $X = \mathbf{R}^d$, if the regions of the partitions are isothetic rectangles (d-dimensional rectangular regions aligned with the coordinate axes), then computing the region containing x is simple.

In the simplest case of learning a scalar-valued function, we introduce a parameter vector,

$$\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle,$$

and a feature vector,

$$\boldsymbol{\phi}(x) = \langle \phi_1(x), \phi_2(x), \ldots, \phi_n(x) \rangle,$$

where

$$\phi_i(x) = \begin{cases} 1 & \text{if } \exists j, 1 \leq j \leq m \wedge \text{Map}_j(x) = i \\ 0 & \text{otherwise} \end{cases}$$

The output of CMAC is defined as the average of the contents of the storage elements determined by the m mapping functions, which is just the quantity,

$$\frac{1}{m} \sum_{i=1}^{m} w_k \Bigg|_{k = \text{Map}_i(x)} .$$

or

$$\frac{1}{m}\mathbf{w}\phi(x).$$

in the case that all m mapping functions determine different storage elements for the input x.

The learning rule for CMAC is just

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \beta\epsilon_t(x_t)\phi(x_t)$$

where the error at time t, $\epsilon(x_t)$, is just the difference between the output of the function we are trying to learn given the training example presented at time t, and the output of CMAC given the same training example,

$$\epsilon_t(x_t) = y(x_t) - \frac{1}{m}\mathbf{w}_t\phi(x_t),$$

assuming here that all m mapping functions determine different storage elements for the input x_t.

The intuition behind this rule is fairly straightforward. Each element, x, of the domain determines m overlapping regions; one from each of the m partitions. Suppose for the sake of argument that these regions map onto m distinct storage elements.[3] These m storage elements will be used to encode the approximate value of $y(x)$, as well as the approximate values of y for the nearby neighbors of x. Elements of the domain that are very near x will likely determine the same m regions, and, hence, the same m storage elements. Elements that are further from x will determine few regions in common with those of x, and hence will have few storage elements in common.

When updating the approximate value of y for x, we will also disturb the approximate values of y for the neighbors of x, but, at least statistically, this disturbance will be in proportion to how near the neighbors are. Very near neighbors will feel the impact of the updates most strongly; more distant neighbors, because they will tend to have fewer storage elements in common with x, will feel it less strongly. Implicit in this method is the assumption that the function we are trying to learn is relatively smooth: if the function varies too much in a given region, then CMAC may not be able to find a good approximation, because CMAC has only a limited amount of storage available to represent the function over the whole domain.

[3] If the hashing function is doing its job correctly, the total number of distinct storage elements determined by the mapping functions for a given input should be a significant fraction of m.

Experiment 4 Apply CMAC to a simple function approximation problem.

The basic idea behind CMAC can be used in a successive refinement strategy to achieve a nice tradeoff between the speed and the accuracy of learning. The strategy is described as follows. Suppose that you want to learn a function, call it y_1. To do so you construct a CMAC system in which the partitions consist of regions which are rather large. This CMAC system will find an approximation to y_1, call it f_1, very quickly, but the approximation is likely to be a poor one, given the coarseness of the mapping. To correct for the inaccuracies of f_1, we build another CMAC system to learn the function, $y_2 = y_1 - f_1$, but this system makes use of partitions consisting of somewhat smaller regions. This second CMAC system will find an approximation to y_2, call it f_2, more slowly than the first CMAC, but it will still represent y_2 more accurately than f_1 represented y_1, and the sum of the two functions, $f_1 + f_2$, will be a better approximation of y_1 than f_1 alone. We can continue in this manner to define a sequence of CMAC systems each using finer partitions than the one before it in the sequence, and each providing a correction for the function corresponding to the sum of functions provided bv the CMAC systems occurring earlier in the sequence.

One way to implement the above strategy is for the learning system to apply each CMAC system in stages, starting with the system using the coarsest partitions and proceeding to those using finer partitions. Each CMAC is run for a fixed number of steps using a learning schedule that tends to zero. This sequential implementation has the disadvantage that it cannot adapt if the function of interest changes slowly over time. An alternative implementation is to run all of the CMAC systems in parallel, using a different fixed learning rate for each CMAC such that the finer the partition the slower the learning (smaller the fixed rate). This parallel approach tends to learn somewhat slower than the sequential approach, but the parallel approach is still quite fast and its ability to adapt to handle time-varying functions makes it useful in a number of applications for which the staged approach would not be effective.

We refer to the general approach of building learning systems using several CMACs employing successively finer partitions as *multi-resolution* CMAC. It turns out that implementing multi-resolution CMAC is actually no more difficult than implementing the version of CMAC described earlier; in some respects it is easier. We describe the basic construction in the following paragraphs.

Suppose that we wish to build a multi-resolution CMAC consisting of

m CMACs with successively finer partitions. Because there are several CMAC's, we need only one partition per CMAC to achieve the redundancy necessary to offset the consequences of hashing collisions. As in the earlier version of CMAC, we assume m partitions and m mapping functions. In the case of multi-resolution CMAC, we require that the partitions are arranged in a sequence so that the ith partition represents a finer partition than the $i-1$ partition.

For the ith CMAC, we define the parameter vector,

$$\mathbf{w}_i = \langle w_{i,1}, w_{i,2}, \ldots, w_{i,n} \rangle,$$

and the feature vector,

$$\boldsymbol{\phi}_i(x) = \langle \varphi_{i,1}(x), \varphi_{i,2}(x), \ldots, \varphi_{i,n}(x) \rangle,$$

where

$$\varphi_{i,j}(x) = \begin{cases} 1 & \text{if } \mathrm{Map}_i(x) = j \\ 0 & \text{otherwise} \end{cases}$$

Each of the m CMACs determines a function,

$$f_i = \mathbf{w}_i \boldsymbol{\phi}_i, \quad \text{for } 1 \leq i \leq m,$$

intended as an approximation to some other function,

$$f_i \approx y_i, \quad \text{for } 1 \leq i \leq m,$$

where y_1 is just the function we have set out to learn, y, and the other $m-1$ functions are defined as follows.

$$y_{i+1} = y_i - f_i, \quad \text{for } 1 \leq i \leq m-1.$$

The output of multi-resolution CMAC is the approximation.

$$y \approx f_1 + f_2 + \cdots + f_m.$$

Learning proceeds simultaneously, using the rules,

$$\mathbf{w}_{i,t+1} = \mathbf{w}_{i,t} + \beta_i \epsilon_{i,t}(x_t) \boldsymbol{\phi}_i(x_t), \quad \text{for } 1 \leq i \leq m,$$

where β_i is the learning rate for the ith CMAC, and the error for the ith CMAC is defined by,

$$\epsilon_{i,t}(x_t) = y_i(x_t) - \mathbf{w}_{i,t} \boldsymbol{\phi}_i(x_t), \quad \text{for } 1 \leq i \leq m.$$

Experiment 5 Apply multi-resolution CMAC to a simple function approximation problem and compare it with the version of CMAC described earlier.

CMAC is a simple, fast, and effective technique for approximating functions. There are more powerful techniques that can solve more difficult problems, but CMAC is a practical method that should be a part of any engineers repertoire of techniques. We rank it alongside the Kalman filter, proportional derivative control, stochastic dynamic programming, and planning by task reduction as useful component techniques for building useful planning and control systems.

The CMAC methods described in this section by no means nullify what was referred to as the curse of dimensionality in Section 9.2. If we have a three-dimensional domain, but the output of the function of interest is independent of, say, the third dimension, then CMAC still has to allocate the storage necessary to represent all three dimensions. In addition, in order to construct a good approximation, CMAC has to sample the three-dimensional space instead of the smaller and completely adequate two-dimensional subspace. An example mentioned earlier illustrates the sort of frustration that can result from this behavior.

Suppose we want a robot to learn a navigation function. The robot has four sensors, a compass or bearing sensor, a position sensor for longitude, a position sensor for latitude, and a light-level sensor. We want the robot to learn a function from the resulting four-dimensional input space to some space of actions. Having taken great pains to teach the robot how to navigate when the light is at one level, we find out that the robot is not able to navigate when the light is at any other level. What we would like is simply to tell the robot to ignore the light level thereby reducing the dimensionality of the learning problem.

What seems easy enough to accomplish in the above example is quite difficult to achieve in general. It is hardly ever the case that one sensor is entirely irrelevant. In most cases, there will be subspaces of the input space that can replaced with spaces of reduced dimensionality. Determining these subspace reductions in dimensionality can be complex, however. In building useful learning systems, the curse of dimensionality will probably always plague us. In lieu of general-purpose solutions, it is hoped that special-purpose techniques will suffice to achieve satisfactory performance for practical problems.

9.4 Rule-Based Learning

In the beginning of this chapter, we introduced learning in terms of approximating functions. The chapter as a whole focusses primarily on learning value functions. In this section, we generalize on this idea of learning value functions to consider a variety of rule-based learning problems.

Value functions are used to derive policies. What we are really interested in learning is optimal policies. All of the techniques that we considered in Section 9.2 can be thought of as attempting to select an optimal policy from a parameterized class of policies. In each case, the parameterized class is represented as a set of rules of the form, if the current state is x, then perform action u, where each rule has an associated parameter or rule strength. In Section 9.2, the rule strengths were just the action values.

This parameterized class of policies is quite simple. Each rule represents a condition/action pair, in which the condition corresponds to the current state of the world and the action corresponds to some control action.

In the following, we generalize to allow rules of the form,

$$\text{If } A_1 \wedge A_2 \wedge \cdots \wedge A_n, \text{ then } C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where the *antecedents*, $\{A_i\}$, and the *consequents*, $\{C_i\}$, are ground atomic formulae in some appropriate representation language. We associate with each such rule a corresponding weight. We could introduce variables to represent rules with quantifiers, but we will not do so here in order to keep the discussion as simple as possible. Neither will we consider the details of any particular representation language though there are some interesting issues with regard to the choice of representation language. Instead, we employ a simple database model for our discussion.

We assume a database consisting of ground atomic formulae. The contents of this database change over time, as determined by the sequence of rules applied and the information provided by the system's sensors. Let Contents(t) denote the contents of the database at time t.

For each rule, r, let Antecedents(r) be the set of antecedents of r, Consequents(r) its consequents, and W(r, t) its weight or strength at time t. We assume an arbitrary *threshold*, $\tau \in \mathbf{R}$, used to determine which rules are applied. A rule, r, is applied at time, t, just in case the following criterion is satisfied,

$$\text{Antecedents}(r) \subset \text{Contents}(t-1) \wedge \text{W}(r, t) > \tau.$$

We will consider some alternative criteria for rule application in just a bit. A rule is said to be *active* at time, t, denoted Active(r, t), just in case it is applied at t. The set of conclusions available at time t is just the union of the consequents of all the rules active at t.

$$\text{Conclusions}(t) = \bigcup_{\text{Active}(r,t)} \text{Consequents}(r).$$

Control actions are initiated using *procedural attachment*. Procedural attachment refers to the practice of associating procedures with the presence or absence of tuples in a relational database or formulae in a predicate-calculus database. In most procedural attachment schemes, there is a program designed to monitor the contents of the database. When a formula is added to or deleted from the database, the monitor program checks to see if there is a procedure associated with the addition or deletion of the formula, and, if so, runs the appropriate procedure.

Finally, we define the contents of the database at t as the union of the sensory information and conclusions available at t,

$$\text{Contents}(t) = \text{Sensors}(t) \cup \text{Conclusions}(t),$$

where Sensors(t) is a set of ground atomic formulae summarizing the data available from the sensors at t.

At each point in time, the rule strengths are updated. For each rule, r, applied at time t, the system performs the following steps, comprising what is generally called the *bucket-brigade* algorithm [6].

1. For each rule, r', active at time $t - 1$ such that

   $$\text{Antecedents}(r) \cap \text{Consequents}(r') \neq \emptyset$$

   update the strength of r' using the following rule,

   $$W(r', t + 1) = \alpha W(r, t),$$

   where $\alpha \in \mathbf{R}$ is a number between zero and one, similar in its use here to the learning rate described in earlier sections.

2. Update the strength of r using the rule,

   $$W(r, t + 1) = W(r, t) - \alpha W(r, t) + R(t),$$

   where R(t) is the reward at time t.

t	0	1	2	3	4	5	6	7	8	9
$W(R1,t)$	100	80	100	100	80	100	100	80	101.6	101.6
$W(R1,t)$	100	100	80	100	100	80	108	108	88	120.8
$W(R1,t)$	100	100	100	110	110	110	172	172	172	197.6
Sensors(t)	$\{A\}$	$\{\}$	$\{\}$	$\{A\}$	$\{\}$	$\{\}$	$\{A\}$	$\{\}$	$\{\}$	$\{A\}$
Contents(t)	$\{A\}$	$\{B\}$	$\{C\}$	$\{A,D\}$	$\{B\}$	$\{C\}$	$\{A,D\}$	$\{B\}$	$\{C\}$	$\{A,D\}$
$R(t)$	0	0	0	60	0	0	60	0	0	60

Table 9.1: Changes in rule strengths over time

For all the rules not applied at t, there is no change,

$$W(r,t+1) = W(r,t).$$

To illustrate the database model and the bucket-brigade algorithm, consider the following simple example.

Let the set of rules be as follows,

R1:	If A, then B,	100
R2:	If B, then C,	100
R2:	If C, then D,	100,

where the number on the far right indicates the rule strength at $t = 0$ in some arbitrary units. Let $\alpha = 0.2$. Suppose that whenever D is added to the database, the system performs an action that is immediately rewarded at a level of 60, employing the same units used for rule strengths. Table 9.1 shows the evolution of the rule strengths for 10 time steps. If the same cycle of sensor input and rewards found in Table 9.1 is allowed to continue indefinitely, the strengths of all three rules will converge to $\frac{60}{\alpha} = 300$. If the rewards are stochastic but average 60, then the rule strengths will never converge but will average 300.

Experiment 6 Provide an example showing how the bucket-brigade algorithm might be applied to the problem of learning to fill tanker trucks, given the flow model described in Chapter 5.

The bucket-brigade algorithm is often used for classification and prediction problems. In classification problems, the system is given a set of features describing its input and asked to assign the input to one of a finite number of categories. For instance, an assembly-line visual inspection system might classify items on a conveyor belt as ready to ship, defective but repairable, or defective and not worth repairing. For the inspection system, the features might correspond to superficial visual attributes, such as the alignment of external parts, or the number and distribution of flaws on a painted surface. In general, not all of the features given to the system will be relevant to making the classification.

In prediction problems, the system is given a set of features describing the state of the system at the current time and asked to predict the state of the system or some particular aspect of the state of the system at some future time. For instance, a system designed to regulate the flow of gas through a commercial pipeline might need to predict transient leaks that prevent the system from delivering gas at the appropriate pressures. In this case, the features might correspond to the current demand for gas, outside temperature, time of day, and pipeline inlet and outlet pressures. The predictions made by the system are used to prevent or reduce the effects of transient leaks by anticipating demand and regulating pipeline inlet pressure.

The simple thresholding method for applying rules descrbed above is not appropriate for most applications. In the case of classification problems where many of the rules correspond to conflicting hypotheses regarding the class of a particular instance, there may be several rules whose strengths are greater than the threshold, but it would not make sense to apply more than one of them to a given input. A similar case arises in control problems in which there are two or more rules vying to set the same parameter to different values. Nor is it generally appropriate only to apply the rule with the greatest strength: parallel rule invocation is often useful in building effective rule-based control systems.

In most practical applications, the decision as to what rule or rules to apply involves criteria in addition to rule strength. In classification problems, the specificity of the rules' antecedents is often taken into account. For instance, using a specificity criterion, given the database, $\{A, B\}$, and the two rules,

R1: If $A \wedge B$, then C, 100
R2: If B, then D, 100,

only the first rule would be applied, since, though both rules have their antecedent conditions satisfied, the first has a more specific antecedent condition than the second.

Most rule application strategies also involve a component of stochastic selection. As we saw in regard to learning optimal policies in stochastic sequential decision problems. the system has to experiment with a variety of rules in order to be assured of finding the optimal one. Similarly, for learning classification and prediction rules, it is necessary to occasionally try rules that are not doing particular well just in case those rules have not as yet had sufficient opportunities to demonstrate their utility.

The bucket-brigade algorithm is often used to select a set of promising rules from a larger set. In rule selection. a set of candidate rules are applied in a set of experiments. their strengths adjusted using the bucket-brigade algorithm. and the subset of rules with rule strengths above a certain threshold are selected as promising. Rule selection addresses just one issue in designing effective learning systems. There is another issue that we have overlooked up until now. This issue concerns where the rules come from.

In sequential decision problems. we are given a set of rules of the form, if the current state is x. then perform u. which can be used to specify all possible policies. Even in this case, the number of such rules is often dauntingly large. In some problems, the number of possible rules is infinite or so large that it is unthinkable to generate and store all of them at once.

There are many techniques for generating new rules given an existing set of rules. Some of them involve methods for generalizing and specializing antecedents and consequents to form new rules. Other techniques use *genetic operators* to construct rules by combining parts of two or more existing rules. A detailed discussion of such techniques is beyond the scope of this chapter. Suffice it to say that effective generation of new rules is an active area of learning research with many open problems. In the last section of this chapter. we provide some references for further reading.

Generally, a complete learning system observes a two-phase cycle of activity. In the first phase, a set of candidate rules is generated using as a basis whatever rules survived the last selection phase. In the second phase. the set of candidate rules is subjected to a series of experiments designed to identify the most useful rules and eliminate the less effective ones. In this chapter, we have focussed primarily on the problem of rule selection. because the corresponding area of research is the best developed and most directly relevant to the problems considered in this book.

9.5 Learning and Observability

In this chapter, we focus on the problem of learning an optimal policy for a stochastic dynamical system with rewards. In some cases, it may be possible to divide the problem into component problems. For instance, if the dynamical system satisfies a separation property, it may make sense to consider two separate learning problems: one concerned with observation, learn how to determine what state you are in, and one concerned with control, learn what action to take given that you know what state you are in. You can divide control still further into system identification, learn a model of the system dynamics and rewards, and regulation, learn an optimal control law given the dynamics and rewards.

In practice, however, breaking the problem into pieces may not be the most effective way to proceed. With regard to observation, you probably do not have to know exactly what state you are in as knowing the proper equivalence class will suffice for some appropriate equivalence relation. With regard to control, for the sort of robotics and automation problems that we are most interested in, observation and control are not separable, in which case the optimal policy for an ideal observer will not be of much use. With regard to identification, as pointed out in earlier sections, it may not be necessary to predict the evolution of the state in order to determine how to act; if we know the value function for a given policy, it is possible to improve that policy without the use of a mode!

In Section 9.2, we considered condition/action rules of the form, if the current state is x, then perform action u. However, for the techniques involving learning action values that we discussed, we might just as well have considered rules of the form, if our perceptions of the current state are y, then perform action u. This assumes, of course, that the set of possible actions includes perceptual actions, otherwise there would be no way for a robot to influence its perception of the current state.

As we mentioned earlier, if the dynamical system is separable, we might try to learn an optimal observer and an optimal policy separately. Alternatively, we might proceed as though there was a one-to-one mapping between the robot's perceptual states and the states of the world. If this actually was the case, then we effectively have an ideal observer since there is no need to know or make use of the mapping from perceptual states to world states.

If such a one-to-one mapping does not obtain, then there will be states of the world that the robot cannot distinguish between using its perceptual apparatus. Perceptual states that map to two or more world states are said to be ambiguous. This ambiguity may not be a problem: there is no need to distinguish between two states if they require the same response. However, if the two states require very different responses, then performance could be adversely affected. There are two problems associated with ambiguity leading to adverse performance. First, how do you detect it, and, second, having detected it what can you do about it.

If you know that the dynamical system is deterministic, then detecting ambiguous perceptual states is rather easy. For a deterministic system, if the perceptual state is unambiguous, the action values, assuming a fixed policy, should converge to fixed values (at least in the limit). However, if the perceptual state is ambiguous, then the action values will vary between those for each of the corresponding world states. Detecting ambiguous perceptual states in a deterministic system can be handled by carefully monitoring the variance in the action values. Detecting ambiguous perceptual states in a stochastic system can be managed with more sophisticated statistical tests.

Once you know that a given perceptual state is ambiguous and that the variance is sufficient to warrant doing something about it, you still have to decide how to deal with the ambiguity. You may be able to simply perform appropriate perceptual actions in order to move to an unambiguous perceptual state. In general, however, this may not be a good idea. For example, it may be that achieving a goal or maximizing a performance index requires that the system pass through perceptually ambiguous states. In general, we recommend simply treating perceptual states as world states, including perceptual actions as possible actions, and using one of the stochastic methods described in Section 9.2. If the dynamical system is deterministic, then it will behave like a stochastic system if there is perceptual ambiguity, but this stochastic behavior will not prevent the system from learning an optimal policy.

Experiment 7 Apply Watkin's stochastic dynamic programming method to learning a navigation function given uncertainty about the robot's position. Assume a Kalman filtering state estimation front end that provides an estimate of the robot's location to serve as input to the control system.

9.6 Further Reading

For more on the update rule of Widrow and Hoff, the perceptron learning rule of Rosenblatt, and discussion of other learning issues consult the text by Nilsson on learning machines [13] or the first part of the text by Duda and

Hart on pattern classification and scene analysis [4]. For an introduction to some of the issues in function approximation, the paper by Poggio and Girosi provides a comparison of a variety of techniques [14].

Our treatment of learning in terms of stochastic decision problems follows that of Barto, Sutton, and Watkins [3]. For more on solving credit-assignment problems in sequential decision tasks, consider the paper by Sutton [17]. The specific method of learning action values considered in this chapter is due to Watkins [19]. The theory of learning automata is also relevant to the issues addressed here and the text by Narendra and Thathachar is an excellent introduction to this area of research [12]. Sutton considers some of the issues involved in combining exploration and prediction to speed learning [18]. Whitehead and Ballard [20] discuss some issues regarding observability in learning to solve sequential decision tasks.

Albus' CMAC method is described in [2]. A multi-resolution CMAC method is analyzed in [11], and the variation on this method suitable for learning time-varying functions is described in [16].

Holland *et al* describe the bucket-brigrade algorithm for credit assignment in rule-based systems [6]. For more on the application of rule-based techniques to problems in planning and control, see Laird *et al* for a general architecture for problem solving [7], Minton *et al* for a perspective that considers certain forms of learning as akin to theorem proving [9], and Mitchell *al* for an approach to learning plans by generalizing past experience [10]. Also see Hammond for a different perspective on learning plans that deviates from the more conventional rule-based approaches [5]. Much of the work on learning plans is related to the work on speedup learning discussed in Chapter 8. Many of the techniques for speedup learning can be characterized in terms of learning to solve problems efficiently by caching the (generalized) solutions to selected problem instances.

Bibliography

[1] Albus, J. S., A Theory of Cerebellar Functions. *Mathematical Biology*, **10** (1971) 25–61.

[2] Albus, J. S., A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, **97** (1975) 270–277.

[3] Barto, Andrew G., Sutton, R. S., and Watkins, C. J. C. H, *Learning and Sequential Decision Making*, Technical Report 89-95, University of Massachusetts at Amherst Department of Computer and Information Science, 1989.

[4] Duda, R. O. and Hart, P. E., *Pattern Classification and Scene Analysis*, (John Wiley and Sons, New York, 1973).

[5] Hammond, Kris, CHEF: A Model of Case-based Planning, *Proceedings AAAI-86, Philadelphia, Pennsylvania*, AAAI, 1986, 267–271.

[6] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R., *Induction: Processes of Inference, Learning, and Discovery*, (MIT Press, Cambridge, Massachusetts, 1987).

[7] Laird, J. E., Newell, A., and Rosenbloom, P. S., SOAR: An Architecture for General Intelligence, *Artificial Intelligence*, **33** (1987) 1–64.

[8] Marr, David, A Theory of Cerebellar Cortex, *Journal of Physiology*, **202** (1969) 437–470.

[9] Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., and Gil, Y., Explanation-Based Learning: A Problem Solving Perspective, *Artificial Intelligence*, **40** (1989) 63–118.

[10] Mitchell. T. M . Utgoff. P.. and Banerji. R.. Learning by Experimentation: Formulating and Generalizing Plans from Past Experience. Michalski. R. S.. Carbonell. J. G.. and Mitchell. T. M.. (Eds.). *Machine Learning: an Artificial Intelligence Approach.* (Tioga. 1983).

[11] Moody. John E.. Fast Learning in Multi-Resolution Hierarchies. Touretsky. David. (Ed.). *Advances in Neural Information Processing*, (Morgan-Kaufmann. Los Altos. California. 1989).

[12] Narendra. Kumpati S. and Thathachar. Mandayam A. L.. *Learning Automata: An Introduction.* (Prentice-Hall. Englewood Cliffs. New Jersey, 1989).

[13] Nilsson. Nils. *Learning Machines*. (McGraw-Hill. New York. 1965).

[14] Poggio. Tomaso and Girosi. Federico. *A Theory of Networks for Approximation and Learning.* Technical Report AI Memo No. 1140. MIT AI Laboratory. 1989.

[15] Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, (Spartan Books, Washington. D.C., 1961).

[16] Shewchuk. John and Dean. Thomas. Towards Learning Time-Varying Functions With High Input Dimensionality. *Proceedings of the Fifth IEEE International Symposium on Intelligent Control. Philadelphia. Pennsylvania*, IEEE, 1990, 383-388.

[17] Sutton, Richard S., Learning to Predict by the Methods of Temporal Differences, *Machine Learning.* **3** (1988) 9–44.

[18] Sutton, Richard S., Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming, *Proceedings 7th International Conference on Machine Learning. Austin. Texas*, 1990.

[19] **Watkins**, C. J. C. H. *Learning from Delayed Rewards*. PhD thesis. Cambridge University, 1989.

[20] **Whitehead**. Steven D. and Ballard. Dana H.. Active Perception and Reinforcement Learning. *Proceedings 7th International Conference on Machine Learning. Austin. Texas*. 1990.

[21] Widrow, B. and Hoff, M. E., Adaptive Switching Circuits. *1960 WESCON Convention Record Part IV. (Reprinted in J. A. Anderson and E. Rosenfeld,* Neurocomputing: Foundations of Research. *The MIT Press, Cambridge, MA. 1988).* 1960. 96–104.